TARTALOMJEGYZÉK

ELŐSZÓ

1 BEVEZETÉS

1.1 RENDSZEREZETT SZÁMíTÓGÉP FELÉPíTÉS 2
1.1.1 Nyelvek, Szintek, és Virtuális Gépek 2
1.1.2 A Jelenlegi Többszintű Gépek 4
1.1.3 A többszintű gépek fejlődése 8

1.2 MÉRFÖLDKŐ A SZÁMÍTÓGÉP ARCHITEKTÚRÁBAN 13
1.2.1 Az nulladik generáció - Mechanikus számítógépek (1642-1945) 13
1.2.2 Az első generáció - Elektroncsövek (1945-1955) 16
1.2.3 A második generáció - Tranzisztorok (1955-1965) 19
1.2.4 A harmadik generáció - Integrált áramkörök (1965-1980) 21
1.2.5 A negyedik generáció - Nagyon nagy mértékű integráltság (1980-?)

1.3 A SZÁMÍTÓGÉPEK VILÁGA 24
1.3.1 Technológiai és Gazdasági Erők 25
1.3.2 A Számítógép választék 26

1.4 PÉLDA SZÁMÍTÓGÉP CSALÁDOKRA 29
1.4.1 A Pentium II bemutatása 29
1.4.2 Az UltraSPARC II bemutatása 31
1.4.3 A picoJava II bemutatása 34

1.5 A KÖNYV ÁTTEKINTÉSE 36

2 SZÁMÍTÓGÉP RENDSZER FELÉPÍTÉSE

2.1 PROCESSZOROK 39
2.1.1 CPU felépítése 40
2.1.2 Utasítás végrehajtása 42
2.1.3 RISC és CISC összehasonlítása 46
2.1.4 A modern számítógépek tervezési elvei 47
2.1.5 Utasítás-szintű párhuzamosság 49
2.1.6 Processzor-szintű párhuzamosság 53

2.2 ELSŐDLEGES MEMÓRIA 56
2.2.1 Bitek 56

2.2.2 Memória címzés 57
2.2.3 Byte elrendezés 58
2.2.4 Hibajavító kódok 61
2.2.5 Közbülső cache memória 65
2.2.6 Memória szervezése és típusai 67

2.3 MÁSODLAGOS MEMÓRIA 68
2.3.1 Memóriák hierarchiája 69
2.3.2 Mágneses lemezek 70
2.3.3 Floppylemez 73
2.3.4 IDE lemez 73
2.3.5 SCSI lemez 75
2.3.6 RAID 76
2.3.7 CD-ROM 80
2.3.8 Az írható CD 84
2.3.9 Az újraírható CD 86
2.3.10 DVD 86

2.4 INPUT/OUTPUT 89
2.4.1 Busz 89
2.4.2 Terminál 91
2.4.3 Az egér 99
2.4.4 A nyomtató 101
2.4.5 A Modem 106
2.4.6 Karakter kódok 109

2.5 ÖSSZEFOGLALÁS 113

3 A DIGITÁLIS LOGIKA SZINTJE

3.1 KAPUK ÉS A BOOL ALGEBRA
3.1.1 Kapu 118
3.1.2 A Boole algebra 120
3.1.3 A Bool-függvények megvalósítása 122
3.1.4 Áramkörök egyenértékűsége 123

3.2 ALAPVETŐ DIGITÁLIS ÁRAMKÖRÖK 128
3.2.1 Integrált áramkörök 128
3.2.2 Kombinált áramkörök 129
3.2.3 Aritmetikai áramkörök 134
3.2.4 Idő 139

3.3 MEMÓRIA 141
3.3.1 Kódnyelv 141
3.3.2 Flip-flop 143

3.3.3 A regiszter 145
3.3.4 A memória felépítése 146
3.3.5 Memória lapka 150
3.3.6 RAM és ROM 152

3.4 CPU LAPKA ÉS A BUSZ 154
3.4.1 CPU lapka 154
3.4.2 Számítógép busz 156
3.4.3 A busz szélessége 159
3.4.4 A busz időzítése 160
3.4.5 A busz eljárás 165
3.4.6 A busz műveletek 167

3.5 PÉLDA CPU EGYSÉGRE 170
3.5.1 A Pentium II 170
3.5.2 Az UltraSPARC II 176
3.5.3 A picoJava II 179

3.6 PÉLDA BUSZOKRA 181
3.6.1 Az ISA Busz 181
3.6.2 A PCI Busz 183
3.6.3 Az Univerzális Soros Busz 189

3.7 CSATLAKOZTATÁS 193
3.7.1 I/O Egységek 193
3.7.2 Cím dekódolása 195

3.8 ÖSSZEFOGLALÁS 198

4. A MIKROFELÉPÍTÉS SZINTJE 203

4.1 MIKROFELÉPÍTÉSI PÉLDA 203
4.1.1 Az adat
4.1.2 Mikroutasítás 211
4.1.3 Mikroutasítás vezérlés: A Mic-1 213

4.2 ISA PÉLDA: IJVM 218
4.2.1 Verem 218
4.2.2 Az IJVM memória modell 220
4.2.3 Az IJVM utasítás készlet
4.2.4 Java fordítása IJVM -mé 226

4.3 MEGVALÓSÍTÁS PÉLDA 227
4.3.1 Mikroutasítás és jelölés 227
4.3.2 ILVM megvalósítása Mic-1 segítségével 232

4.4 A MIKROFELÉPÍTÉS SZINTJÉNEK TERVEZÉSE 243

4.4.1 A gyorsaság és a költségek ellentmondása 243
4.4.2 A végrehajtás útjának lerövidítése 245
4.4.3 Előbetöltést alkalmazó tervezés: A Mic-2
4.4.4 Csővezetékes tervezés: A Mic-3 253
4.4.5 Hét-szakaszos csővezeték: A Mic-4 260

4.5 TELJESÍTMÉNY JAVÍTÁSA
4.5.1 Közbülső memória 265
4.5.2 Elágazás becslése 270
4.5.3 Sorrenden kívüli végrehajtás és regiszter újracimkézése 276
4.5.4 Mérlegeléses végrehajtás 281

4.6 PÉLDÁK A MIKROFELÉPÍTÉS SZINTJÉRE 283
4.6.1 A Pentium II CPU mikrofelépítése 283
4.6.2 Az UltraSPARC II CPU mikrofelépítése 288
4.6.3 A picoJava II CPU mikrofelépítése
4.6.4 A Pentium az UltraSPARC és a picoJava összehasonlítása 296

4.7 ÖSSZEFOGLALÁS 298

5 AZ UTASÍTÁS KÉSZLET FELÉPÍTÉSÉNEK SZINTJE (ISA) 303

5.1 AZ ISA SZINT ÁTTEKINTÉSE 305
5.1.1 Az ISA szint tulajdonságai 305
5.1.2 Memória modellek 305
5.1.3 Regiszterek 309
5.1.4 Utasítások 311
5.1.5 A Pentium II ISA szintjének áttekintése 311
5.1.6 Az UltraSPARC II ISA szintjének áttekintése 313
5.1.7 A Java Virtuális Gép áttekintése 317

5.2 ADAT TÍPUSOK
5.2.1 Numerikus adat típusok 319
5.2.2 Nem numerikus adat típusok 319
5.2.3 A Pentium II adat típusai 320
5.2.4 Az UltraSPARC II adat típusai 321
5.2.5 A Java Virtuális Gép adat típusai 321

5.3 UTASÍTÁS FORMÁTUMOK 322
5.3.1 Az utasítás formátumok tervezési jellegzetességei 322
5.3.2 Utasítás kódok kiterjesztése 325
5.3.3 A Pentium II utasítás formátuma 327
5.3.4 Az UltraSPARC II utasítás formátuma 328
5.3.5 A JVM utasítás formátuma 330

5.4 CÍMZÉS 332
5.4.1 Címmegadási módszerek 333
5.4.2 Azonnali címzés 334
5.4.3 Közvetlen címzés 334
5.4.4 Regiszter címzés 334
5.4.5 Regiszter indirekt címzés 335
5.4.6 Indexelt címzés 336
5.4.7 Alap-indexelt címzés 338
5.4.8 Verem címzés 338
5.4.9 Az elágazó utasítások címmegadási módszerei 341
5.4.10 Az utasításkódok és a címzési módok merőlegessége 342
5.4.11 A Pentium II címmegadási módszerei 344
5.4.12 Az UltraSPARC II címmegadási módszerei 346
5.4.13 A JVM címmegadási módszerei 346
5.4.13 A címmegadási módszerek áttekintése 347

5.5 UTASÍTÁS TÍPUSOK 348
5.5.1 Adat mozgató utasítások 348
5.5.2 Kétcímű utasítások 349
5.5.3 Egycímű utasítások 350
5.5.4 Összehasonlító és feltételes elágazó utasítás 352
5.5.5 Eljáráshívó utasítások 353
5.5.6 Ciklusvezérlés 354
5.5.7 Input/Output 356
3.5.8 A Pentium II utasításai 359
5.5.9 Az UltraSPARC II utasításai 362
5.5.10 A picoJava II utasításai 364
5.5.11 Az utasításkészletek összehasonlítása 369

5.6 AZ IRÁNYÍTÁS FOLYAMATA 370
5.6.1 Egymást követő vezérlések és eljárások 371
5.6.2 Eljárásmódok 372
5.6.3 Korutinok 376
5.6.4 Csapdák 379
5.6.5 Megszakítások 379

5.7 EGY RÉSZLETES PÉLDA: HANOI TORNYAI 383
5.7.1 Hanoi tornyai a Pentium II Assembly nyelvben 384
5.7.2 Hanoi tornyai az UltraSPARC II Assembly nyelvben 384
5.7.3 Hanoi tornyai a JVM Assembly nyelvben 386

5.8 AZ INTEL IA-64 388
5.8.1 A Pentium II problémája 390
5.8.2 Ai IA-64 Mocell: Egyértelműen meghatározott párhuzamos utasítások 391
5.8.3 Előjelzés 393
5.8.4 Mérlegeléses betöltés 395
5.8.5 A valóság vizsgálata 396
5.9 ÖSSZEFOGLALÁS 397 OLDAL
5.10

***[1-5]

BEVEZETÉS

A digitális számítógép egy olyan gép, ami problémákat tud megoldani az embereknek az általuk kiadott utasítások alapján. Az utasítások sorrendjét, mely leírja, hogyan kell végrehajtani egy adott feladatot, PROGRAMnak nevezünk. Minden egyes számítógép elektronikus áramköre képes felismerni és közvetlenül végrehajtani egyszerű utasítások meghatározott sorozatát. Minden programot a végrehajtás előtt ilyen egyszerű utasításokká kell alakítani. Ezek az alapvető utasítások ritkán bonyolultabbak, mint például az, hogy

- adj össze két számot
- ellenőrizd, hogy a szám nulla-e
- másold át az adathalmaz egy részét a számítógép memóriájának egyik részéből a másikba

Együttvéve a számítógép egyszerű utasításai alakítják azt a nyelvet, mely segítségével tudnak az emberek kommunikálni a számítógéppel. Ezt a nyelvet GÉPI NYELVnek hívjuk. Az új számítógépeket tervező embereknek el kell dönteniük, hogy milyen utasításokat tartalmazzon az új számítógép gépi nyelve. Azért, hogy a szükséges elektronikai berendezések bonyolultságát és árát csökkentsék, általában a lehető legegyszerűbb utasításokat próbálják elkészíteni a tervezők figyelembe véve, hogy mire akarják majd használni a gépet illetve milyen teljesítménybeli követelménynek kell megfelelniük. A legtöbb gépi nyelv nagyon egyszerű, ezért van az, hogy az emberek számára bonyolult és fárasztó dolgozni velük.

Ez az egyszerű megfigyelés idővel oda vezet, hogy egy számítógép tervezése, felépítése egy sor absztrakcióra (elvonatkoztatásra) épül, és minden egyes absztrakció egy másikra. Ily módon a komplexibilitás (bonyolultság) megérthető, megismerhető, és számítógépes rendszerek tervezhetők módszeres, rendezett módon. Ezt a megközelítést STRUKTURÁLT SZÁMÍTÓGÉP SZERKEZETnek hívjuk és a könyvben később is így használjuk majd. A következő fejezetben leírjuk, mit értünk ezen a fogalmon. Azután pedig néhány történelmi fejlesztést, állapotot és fontosabb példákat érintünk.

1.1 STRUKTURÁLT SZÁMÍTÓGÉP SZERKEZET

Ahogy fent említettük, nagy távolság van az emberek számára kényelmes és a számítógépek számára kényelmes dolgok között. Az emberek “X”-et akarnak csinálni, a számítógépek viszont csak “Y”-ra képesek. Ez okozza a problémát. Ennek a könyvnek a célja az, hogy megmagyarázza, hogyan is oldható meg ez a probléma.

1.1.1 NYELVEK, SZINTEK, VIRTUÁLIS (LÁTSZÓLAGOS) GÉPEK

A probléma két oldalról közelíthető meg: mindkettő egy új utasítás sorozat tervezését foglalja magába, melynek használata az emberek

számára kényelmesebbek, mint a beépített gépi utasításoké. Ezek az új utasítások szintén alkotnak egy nyelvet, melyet L1-nek (Level

1, 1. szint) hívunk, csakúgy, mint a beépített utasítások, melyek nyelvét L0-nak hívunk. A két megközelítés abban különbözik egymástól, hogy az L1 nyelven megírt programokat milyen módon hajtaja végre a számítógép, mely persze csak a saját, Lo nyelvén megírt programként tud értelmezni.

Az L1 nyelven megírt program elvégzésének egyik módszere, hogy először minden utasítást helyettesítünk egy megfelelő L0 nyelvű utasítás sorozattal. Az így keletkezett program csak L0 nyelvű utasításokból áll. A számítógép ezután ezt az új L0 nyelvű programot végzi el az eredeti L1 helyett. Ezt a technikát FORDÍTÁSnak nevezzük.

A másik eljárás szerint L0 nyelven íródik meg egy program, mely az L1 nyelvű programot bemenő adatként kezel és végrehajtja úgy, hogy minden egyes utasítást sorban megvizsgál és közvetlenül végrehajtja az L0 nyelvű megfelelő utasításokat. Ezen eljárás során nem szükséges létrehozni egy új programot L0 nyelven. Ezt a módszert TOLMÁCSOLÁSnak hívjuk, a program, amely ezt végzi a TOLMÁCS.

A fordítás, a tolmácsolás hasonló eljárás. Mindkét módszer során az L1 nyelven megírt utasítások elvégzése végül a megfelelő L0 nyelvű utasítások elvégzésével történik meg. A különbség az, hogy a fordítás során az eredeti L1 program először egy L0 programmá lesz alakítva, majd az L1 programot tovább nem is figyelve, az új L0 program töltődik be a számítógép memóriájába és ezt hajtja végre a gép. A végrehajtás során ez az újonnan létrehozott L0 program fut és van a számítógép irányítása alatt.

A tolmácsolás során minden egyes L1 utasítás-megvizsgálás és dekódolás után rögtön végrehajtásra kerül. Lefordított program nem is készül. Itt a tolmács-program az, akinek az irányítása alatt van a számítógép.

Számára az L1 program csak adat. Mindkét módszert, és egyre inkább a kettő kombinációját, széles körben használják.

Ahelyett, hogy a fordítás és a tolmácsolás definícióján (meghatározásán) rágódunk, gyakran sokkal egyszerűbb elképzelni egy feltételezett számítógép vagy virtuális gép létezését, mely gép nyelve L1. Nevezzük ezt a virtuális gépet M1-nek (és az L0-nak megfelelő virtuális gépet M0-nak). Ha egy ilyen gépet elég olcsón lehetne előállítani, akkor egyáltalán nem lenne szükség L1-re vagy olyan gépre, mely L1 nyelven végzi a programokat. Az emberek egyszerűen megírnák a programjaikat L1 nyelven és a számítógép közvetlenül elvégezné. Annak ellenére, hogy a VIRTUÁLIS GÉP, melynek nyelve L1, túl drága vagy túl bonyolult lenne elektronikus áramkörökből kialakítani, ezért számára az emberek megírhatják a programot. Ezeket a programokat vagy tolmácsolhatjuk vagy lefordíthatjuk egy L1-ben írt programmal, mely maga is elvégezhető egy létező számítógéppel. Más szóval, az emberek ugyanúgy írhatnak programokat a virtuális gépekre, mintha valójában is léteznének.

Ahhoz, hogy a tolmácsolás vagy fordítás alkalmazható legyen, az L0 és L1 nyelveknek nem szabad "túl" különbözőnek lennie. Ez a kényszer gyakran azt jelenti, hogy az L1 bár jobb, mint a L0, messze van az ideálistól a legtöbb esetben (alkalmazásnál). Ez valószínűleg ellentmond az L1 nyelv létrehozásának alapelvének, melynek célja, hogy megkönnyítse a programozóknak az algoritmusok kifejezését egy olyan nyelven, amely jobban megfelel a gépeknek, mint az embereknek. Ennek ellenére a helyzet nem reménytelen.

A nyilvánvaló megoldás az, hogy találjunk ki még egy utasítás sorozatot, ami még felhasználó-orientáltabb, és kevésbé gép-központú, mint az L1. Ez a harmadik utasítás sorozat szintén alkot egy nyelvet, melyet nevezzünk L2-nek (a virtuális gépet pedig M2-nek). Az emberek írhatnak programokat L2-ben úgy, mintha egy virtuális gép L2 gépi nyelvvel valójában létezne. Ezek a programok lefordíthatók L1 nyelvre, vagy egy L1 nyelven írt tolmács programmal végrehajthatók.

Egész sor, az előzőeknél mindig jobb és jobb új nyelvek kifejlesztése akár a végtelenségig is folytatódhat, míg egy valóban megfelelő el nem készül. Minden egyes nyelv az előző nyelvet, mint alapot használja, így úgy tekinthetjük az ezt a technikát használó számítógépet, mint RÉTEGek vagy SZINTek összességét, ahogy az

(1-1)-es ábra is mutatja. A legalsó szint vagy nyelv a legegyszerűbb, a legfelső a legbonyolultabb.

A virtuális gép és a nyelv között egy nagyon fontos összefüggés (kapcsolat) van. Minden gépnek van egy gépi nyelve, mely tartalmazza az összes utasítást, melyet a számítógép el tud végezni. Tehát minden gép meghatároz egy nyelvet. Hasonlóan a nyelv meghatározza a gépet – azaz azt a gépet, amely az adott nyelven írt programok összes utasítását véghez tudja vinni. Természetesen, egy adott nyelv által meghatározott gép lehet, hogy túlságosan bonyolult és drága lenne elektromos áramkörökből felépíteni, ennek ellenére azért el lehet képzelni. Egy gép, melynek gépi nyelve C++ vagy COBOL, valóban nagyon komplex, de a mai technológiával könnyen elkészíthető. Azonban van egy jó indok, hogy miért ne építsünk ilyen számítógépeket: nem kifizetődő a többi technológiához viszonyítva.

Számítógép, mely n szinttel rendelkezik, n különböző virtuális gépnek tekinthető, mindegyik külön saját gépi nyelvvel. A “szint” és a “virtuális gép” definícióját felcserélhetően használjuk. Csak az L0 nyelven írt programok végezhetők el közvetlenül elektromos áramkörökkel tolmácsolás vagy fordítás nélkül. Az L1, L2, … Ln nyelveken írt programokat vagy egy alacsonyabb szinten futó tolmácsprogrammal tolmácsolni kell vagy lefordítani egy alacsonyabb szintnek megfelelő nyelvre.

Azoknak az embereknek, akik programokat írnak n szintű virtuális gépekre, ismerniük kell az alap fordító és tolmácsoló programokat. A gép felépítése biztosítja, hogy ezek a programok valahogy végre lesznek hajtva. Az nem lényeges, hogy egy tolmács-program végzi el az utasításokat lépésről lépésre, mely tolmács-programot egy másik tolmács-program hajtja végre, vagy közvetlenül az elektromos berendezések végzik el a feladatokat. Mindkét esetben azonos eredményt kapunk: a programok lefuttatódnak.

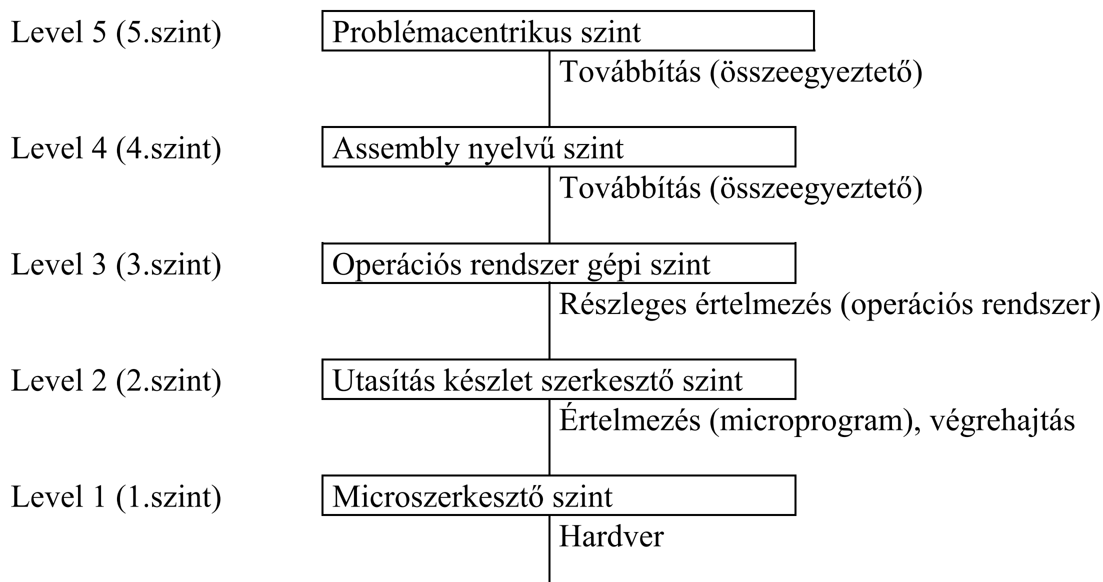
A legtöbb n-szintű gépet használó programozót csak a felső szint érdekli, az, amelyik a legkevésbé közelíti meg a gépi nyelvet a szintek legalján. Azonban, ha az emberek meg szeretnék érteni, hogyan működnek valójában a számítógépek, meg kell ismerniük az összes szintet. Azoknak az embereknek, akik új számítógépek vagy új szintek (pl.: új nyelvek) készítése, tervezése iránt érdeklődnek, szintén ismerniük kell nemcsak a legfelső szintet, hanem a többi, alacsonyabb szinteket is. A számítógépek, mint különböző szintek sorozatából felépülő gépek, tervezési módszerei, koncepciói, a különböző szintek részletei, leírásai adják ennek a könyvnek a fő témáját.

Level n (n. szint)	**Mn. virtuális gép, n. szintű gépi nyelvvel**
Level 3 (3.szint)	M3. virtuális gép, 3. szintű gépi nyelvvel
Level 2 (2. szint)	M2. virtuális gép, 2. szintű gépi nyelvvel
Level 1 (1.szint)	M1. virtuális gép, 1. szintű gépi nyelvvel
Level 0 (0. szint)	M0. aktuális számítógép, 0. szintű gépi nyelvvel

1-1. ábra : A több-szintű gép

1.1.2 NAPJAINK TÖBB-SZINTŰ SZÁMÍTÓGÉPEI

A legtöbb modern számítógép két vagy több szintből áll. Léteznek hat vagy több szintű gépek is, a (1-2). ábra legalján található a gép valódi hardvere. Ennek az áramkörein átközvetítődik az első szint gépi nyelve. A teljesség kedvéért meg kell említenünk egy másik szintet is, amely a mai 0-s szintünk alatt helyezkedik el. Ez a szint nincs feltüntetve az ábrán, mivel ez már az elektromérnökök szakterülete (és ezért ez a könyv nem is tér ki rá). A neve: ESZKÖZTÁR. Ezen a szinten a tervező külön kezeli az egyes tranzisztorokat, melyek a számítógép-tervezők a legegyszerűbb szintű egységek. Ha megkérdeznénk, hogyan működik egy tranzisztor, az egyenesen a térfizikához vezet minket.



Level 0 (0.szint) | Digitális logika szintje

1-2. ábra : A hat-szintű számítógép

A legalsó szinten, melyet tanulmányozni fognak a DIGITÁLIS LOGIKA SZINTjén, a tárgyakat KAPUKnak hívják, bár analóg alkatrészekből épülnek fel, mint például: tranzisztorok, a "kapuk"-at pontosan lehet modellezni, ábrázolni, mint digitális eszközöket. Minden "kapu"-nak van egy vagy több digitális bemenete (jelek, melyek 1-et vagy 0-át jelenítenek meg) és mint kimenetek ezen bemenetek, olyan egyszerű műveleteket végeznek el, mint az ÉS vagy a VAGY. Minden egyes "kapu" maximum egy maréknyi tranzisztorból áll. Néhány kapu összerakható úgy, hogy 1 bit memóriát alkosson, mely egy 0-ást vagy egy 1-est képes tárolni. Az ilyen 1-bites memóriákat 16-os, 32-es és 64-es csoportokon lehet rendezni, hogy adattárolót alkossunk. Minden ilyen ADATTÁROLÓ egy 2-es (bináris) számrendszerbe tartozó számot tud tárolni bizonyos felső korláttal. A "kapu"-kat úgy is össze lehet illeszteni, hogy a számító-, számoló-egység magvát alkossák. A "kapu"-kat és a digitális logikai szintet a harmadik fejezetben tárgyaljuk.

A következő szint a MIKROÖSSZEÁLLÍTÁSI SZINT, ezen a szinten általában 8 és 32 közötti az a szám, melyek az adattároló csoportokat alkotják, melyek a helyi memóriát és egy logikai áramkört képeznek. Bizonyos **ALU**-t (ARITHMETIKAI LOGIKAI EGYSÉG), melyek képesek végrehajtani egyszerű arithmetikai logikát.

***[6-9]
Nem érkezett meg. …Tibor: h938527

***[10-13]

Tóth Éva,10-11.oldal

6. A program végrehajtása megkezdődött.Elég gyakran megesett,hogy nem működött és váratlanul félbeszakadt.Rendszerint a programozó babrált a konzol kapcsolókkal és későn vette észre a konzolfényt.Ha szerencséje volt rájött a hibára és kijavította azt,majd visszatért a kabinethez,ami tartalmazta a nagy FORTAN szerkesztőt,melynek segítségével mindent kezdhetett elölről.Ha kevésbé volt szerencséje kapott a memória tartalmáról egy kiírást,ún.**core dump**-ot,amit végül otthon áttanulmányozhatott.

Ez az eljárás,kisebb változtatásokkal évekig elfogadott volt a számítógép központokban. Rákényszerítette a programozókat,hogy megtanulják hogyan működtessék a gépet és tudják mit tegyenek,amikor elromlik,ami elég gyakran megesett. A gép sokszor állt kihasználatlanul,miközben a szakemberek kártyákkal próbálták megoldani a problémát vagy azon törték a fejüket,vajon a programok miért nem működnek megfelelően.

1960 körül megpróbálták csökkenteni az elvesztegetett idő mennyiségét az operátor munkájának automatizálásával. Az operációs rendszernek nevezett programot állandó jelleggel a számítógépen tárolták.A programozó bizonyos vezérlőkártyákról gondoskodott a programmal együtt, hogy az olvasható és végrehajtható legyen az operációs rendszer segítségével.Az 1-3.ábra egy kártyacsomag(deck) mintáját mutatja be, mely az egyik széles körben elterjedt operációs rendszer, FMS (FORTRAN Monitor System), Az IBM 709-ből.

```
                  *JOB,5494,BARBARA
                  *XEQ
                  *FORTRAN
 FORTRAN program {
                  *DATA
      Data kártyák {
                  *End
```

1-3.ábra Az FMS operációs rendszer egy minta feladata

Az operációs rendszer leolvasta a *JOB kártyát és információt könyvelése céljából használta fel azt. (A csillagjelet vezérlőkártyák azonosítására használták azért,hogy ne tévesszék össze a program és adat kártyákkal.) Ezután a *FORTRAN kártyát olvasta le,mely a FORTRAN szerkesztő számára volt utasítás mágneses szalagról történő betöltéséhez.Ezt követően a szerkesztő beolvasta és megszerkesztette a FORTRAN programot.Amikor a szerkesztés befejeződött visszatért leellenőrizni az operációs rendszert,mely ezután a *DATA kártyát olvasta le.Ez utasítás volt fordító program végrehajtásához,mely használva a kártyákat követte a *DATA kártyát,mint adatot.

Bár az operációs rendszert úgy tervezték ,hogy automatizálja az operátor munkáját (innen származik a név) ,az első lépés volt egy új virtuális gép fejlődésében.A *FORTRAN kártya virtuális “szerkesztő program” utasításnak,míg a *DATA kártya egyszerűen,virtuális “végrehajtó programnak” tekinthető.A mindössze két utasítással rendelkező szint nem volt túl magas,de ez csak kiindulópont ebbe az irányba.

A rákövetkező években az operációs rendszerek egyre bonyolultabbá váltak. Új parancsokkal ,lehetőségekkel és jellegzetességekkel látták el az ISA-szintet,mígnem egy új szint körvonele kezdett kirajzolódni.Ezen új szint parancsai közül néhány azonos volt az ISA-szint parancsaival,míg mások,különösen az input/output parancsok teljesen különböztek tőlük.Az új parancsokat gyakran **operating system macro**-nak vagy **supervisor call**-nak hívták.A szokásos szakkifejezés ma a **system call**.

Az operációs rendszerek más irányokba is fejlődtek.Az elsők leolvasták a kártyacsomagokat és az outputot soros nyomtatóval nyomtatták ki.Ezt az elrendezést **batch system**-nek (kötegelt rendszer) nevezték. Általában több óra várakozási idő telt el ,mire a program engedelmeskedett és az eredmények elkészültek. Ilyen körülmények között igen bonyolult volt a szoftver fejlesztése.

Az 1960-as évek elején a Dartmouth College,MIT, kutatói és mások kifejlesztették az operációs rendszereket úgy,hogy engedélyezték a (multi) programozóknak a számítógéppel történő közvetlen kommunikálást.Ezekben a rendszerekben kapcsolták össze telefonvonalakon keresztül a remote terminálokat a központi számítógéppel.A CPU-t sok felhasználó között osztották meg.A programozók begépelhették a programot és majdnem azonnal visszakapták a begépelt

Tóth Éva,11-13.oldal

eredményeket az irodában,az otthoni garázsban vagy bárhol,ahol a terminált elhelyezték.Ezeket a rendszereket hívják **timesharing system**-nek (időosztott rendszer).

Számunkra az operációs rendszerek azon részei érdekesek,melyek a 3.szintben jelen levő parancsokat és jellemvonásokat interpretálják,de az ISA-szintben nem találhatók meg még az időosztás szempontjából sem. Bár nem fogjuk kihangsúlyozni,mégsem szabad megfeledkezni arról ,hogy az operációs rendszerek többet csinálnak,minthogy sajátságokat interpretálnak,melyeket az ISA-szinthez adtak hozzá.

A feladatkör átváltozása mikrokódba

Egyszercsak a mikroprogramozás egyszerűvé vált (1970-re), a tervezők rájöttek arra,hogy mindössze a mikroprogran kiterjesztésével képesek új utasításokat hozzákapcsolni. Más szavakkal, képesek "hardvet" (azaz új gépi utasításokat) programozással hozzáadni.

Ez a felfedezés "virtuális robbanáshoz" vezetett a gépi utasítású készülékek terén, amikor a tervezők egymással versengve hoztak létre nagyobb és jobb utasítású gépeket.Ezen parancsok közül sok nem volt lényeges,abban az értelemben,hogy eredményük könnyebben megvalósítható volt létező parancsokkal, de gyakran némileg gyorsabbak voltak,mint a létező parancsok sorozata. Például sok gépnek volt egy INC (INCrement) nevű utasítása,mely minden számhoz hozzáadaott egyet. Amióta ezek a gépek rendelkeznek az általános ADD utasítással is ,ami olyan speciális parancs, mely hozzáad egyet (vagy akár 720-at) szükségtelenné vált.Bárhogyanis, az INC általában kicsit gyorsabb volt, mint az ADD , így ezt alkalmazták inkább.

Sok egyéb utasítást is hozzáadtak a mikroprogramhoz hasonló okokból.Ezek gyakran tartalmaztak:

1.Egész értékű szorzásra és osztásra való parancsokat.

2.Lebegőpontos aritmetikai utasításokat.

3.Műveletekbe való hívásra és műveletekből való visszatérésre vonatkozó parancsokat.
4.Kapcsolás felgyorsítására vonatkozó utasításokat.
5.Karakterláncok kezelésére való utasításokat.

Ráadásul egyszercsak a géptervezők rájöttek , hogy milyen egyszerű új utasítások hozzáadása, így elkezdtek újabb tulajdonságok után kutatni , amiket hozzáadhattak saját mikroprogramjukhoz.Néhány példa arra,hogy mit tartalmaznak ezek a kiegészítések:
1.Tulajdonságokat,melyek felgyorsítják a számításokat tartalmazva azok sorrendjét (megjelölés és közvetett címzés)
2.Tulajdonságokat,melyek engedélyezik a programok memóriába történő átmozgatását, miután elindították őket (áthelyezési lehetőségek)
3.Félbeszakító rendszerek, melyek jelzik a számítógépnek,amint egy input/output művelet befejeződött
4.Egy program megszakításának és egy másik elindításának lehetősége kevés utasítással (eljárás kapcsolás)

Sok más tulajdonságot és lehetőséget is hozzáadtak az évek során, néhány különleges tevékenység felgyorsításának érdekében.

A mikroprogramozás kiküszöbölése

A mikroprogramozás virágzó korszakában nagyon elszaporodtak a mikroprogramok (1960-as és 1970-es évek). A probléma csak az volt ,hogy elterjedésükkel egyidejűleg egyre lassabbá váltak. Végül náhány kutató rájött,hogy a mikroprogram kiküszöbölésével , az utasítássorozat leredukálásával és a megmaradó parancsok közvetlen végrehajtásával (azaz a hardver felügyeli az adatok útvonalát) fel lehet gyorsítani a gépeket. Bizonyos értelemben a számítógép terv teljes kört tett meg , majd visszakanyarodott, mielőtt Wilkes elsőként feltalálta a mikroprogramozást.

Tóth Éva,13.oldal

A vita tárgy megmutatni , hogy a hardver és a szoftver közötti határvonal korlátlan és állandóan változik.A mai szoftver lehet, hogy a holnapi hardver, és viszont.Továbbá a különböző szintek
közötti határok is változatlanok még. A programozó szemszögéből az,hogy egy utasítás hogyan megy végbe tulajdonképpen lényegtelen (kivéve talán a sebességet).Az ISA-szinten programozó használhatja a sokszoros parancsot , mintha hardver parancs lenne anélkül,hogy aggódnia kellene felőle, vagy tudatában lenni annak , hogy vajon valóban hardver utasítás-e. Ami az egyik személynek hardver az a másiknak szoftver.Mindezen témára még vissza fogunk térni ebben a könnyvben.

1.2. Mérföldkövek a számítógép architektúrában

A modern digitális számítógép fejlődése során százával terveztek és építettek különböző típusú számítógépeket.A legtöbb már rég feledésbe merült,de néhány közülük lényeges hetással volt a modern eszmékre.Ebben a fejezetben röviden vázoljuk a történelmi fejlődés kulcsfontosságú lépését annak érdekében, hogy jobban megértsük, hogyan jutottunk el a jelenlegi állapotokhoz. Szükségtelen mondani, hogy ez a fejezet csak felületesen érinti a kiemelkedő mozzanatokat és sok kérdést nyitottan hagy.Az 1-4.ábra néhány mérföldkő jelentőségű gépet sorol fel , melyeket ebben az időszakban fejlesztettek ki.A Slater (1987) című könyv sok jó információt ad további történelmi jelentőségű személyekről, akik megalapozták a számítógép korszakát.A számítógép korát megalapozó személyekről szóló,Morgan által gazdagon illusztrált könyvben (1997) a rövid életrajzokat és szép színes képeket Louis Fabian Bachrach készítette.Másik életrajzi könyv pedig a Slater (1987).

1.2.1. A Nulladik Generáció-Mechanikus Számítógépek (1642-1945)

Az első személy, aki működő számítógépet konstruált, a francia tudós, Blaise Pascal (1623-1662) volt,akinek a tiszteletére adták a programnyelvnek a Pascal nevet.Ezt a készüléket Pascal 1642-ben, 19 éves korában készítette azért, hogy édesapjának segítsen, aki adóbeszedő volt a francia kormánynál.A szerkezet teljesen mechanikus volt , melyet fogaskerekek működtettek és kézi hajtású indítókarral lehetett üzembe helyezni.

Pascal gépe csak összeadni és kivonni tudott, de harminc évvel később a nagyszerű német matematikus, Baron Gottfried Wilhelm von Leibniz (1646-1716) konstruált egy másik mechanikus gépet, mely már szorozni és osztani is tudott.Ennek következményeként Leibniz három évszázaddal ezelőtt megalkotta a mai értelemben vett négy-műveletes zsebszámológép megfelelőjét.

150 évig semmi nem történt, amikoris a University of Cambridge matemetikus professzora, Charles Babbage (1792-1871), a sebességmérő feltalálója megtervezte és megépítette a **difference engine**-t. Ezt a mechanikus készüléket, mely a Pascal-éhoz hasonlóan csak összeadni és kivonni tudott, úgy tervezte, hogy kiszámolja a számtáblákat, melyek igen hasznosak a tengerészeti kormányzásnál.

***[14-17]
14.old.

Év	Név	Készítette	Megjegyzés
1834	Analitikai motor	Babbage	Első kísérlet digitális számítógép készítésére
1936	Z1	Zuse	Az első relékkel működő számítógép
1943	COLOSSUS	A brit kormány	Az első elektronikus számítógép
1944	Mark I	Aiken	Első amerikai általános rendeltetésű számítógép
1946	ENIAC I	Eckert/Mauchley	Ekkor kezdődik a modern számítástechnika története
1949	EDSAC	Wilkes	Első belső programvezérlésű számítógép
1951	Whirlwind	M.I.T.	Az első real-time számítógép
1952	IAS	Von Neumann	A ma használatban lévő számítógépek is ezen az elven alapszanak
1960	PDP-1	DEC	Az első miniszámítógépek (50 eladott példány)
1961	1401	IBM	Hallatlanul népszerű kis üzleti gép
1962	7094	IBM	A 60-as évek elején a tudományos számítások kerülnek túlsúlyba
1963	B5000	Burroughs	Az első magas szintű programozási nyelvet biztosító számítógép
1964	360	IBM	Az első sorozatban gyártott számítógép (számítógépcsalád)
1964	6600	CDC	Az első tudományos szuperszámítógép
1965	PDP-8	DEC	Az első tömegesen elterjedt miniszámítógép (50000 eladott példány)
1970	PDP-11	DEC	A 70-es évek elejére a miniszámítógépek válnak uralkodóvá
1974	8080	Intel	Az első általános rendeltetésű chipenként 8 bites számítógép
1974	CRAY-1	Cray	Első párhuzamos-feldolgozású szuperszámítógép
1978	VAX	DEC	Az első 32 bites szuperszámítógép
1981	IBM PC	IBM	Ekkor kezdődik a modern PC-k kora
1985	MIPS	MIPS	Az első reklám RISC-gép
1987	SPARC	Sun	Az első SPARC-alapú RISC munkaállomás
1990	RS6000	IBM	Az első szuperskalár számítógép

1.4 ábra.: A modern digitális számítógépek fejlődésének néhány mérföldköve

A gép teljes megépítése azon alapult, hogy egy egyszerű algoritmust futtatott, a “véges differenciálás” eljárását polinomokat használva. A differenciálmotor legérdekesebb vonása a kiviteli eljárása volt: az eredményeket egy réz gravírozott lemezbe ütötte egy acél tűvel, ezzel előrevetítve a későbbi egyszer írható információhordozókat, mint pl. a lyukkártya és a CD-ROMok.

Bár a differenciálmotor meglehetősen jól működött, Babbage gyorsan ráunt a gépre, amely mindössze egy algoritmust tudott futtatni. Elkezdte egyre több idejét és családja vagyonának (nem beszélve a 17000 fontról a kormány pénzéből) nagy részét arra áldozni, hogy megtervezze és megépítse az analitikai motornak nevezett utódot. Az analitikai motornak négy alkotóeleme volt: a tároló (memória), a központi egység (számítást végző egység), a bemeneti egység (lyukkártya-olvasó) és a kimeneti

egység (a lyukasztott és a nyomtatott adatok).

15.old.

A memória 50 tízes számrendszerbeli számjegyű szóból állt, és mindegyik változókat és eredményeket tartalmazott. A központi egység elő tudta hívni az operanduszokat a tárolóból, azután összeadta, kivonta, szorozta vagy elosztotta őket egymással , majd az eredményt visszajuttatta a tárolóba. A differenciálmotorhoz hasonlóan ez is teljesen mechanikus volt.

Az analitikai motor óriási előrelépését az jelentette, hogy általános rendeltetésű volt. Beolvasta az utasításokat a lyukkártyáról, és végrehajtotta azokat. Néhány parancs arra utasította a gépet, hogy hívjon elő 2 számot a tárolóból, vigye őket a központi egységbe, végezze el rajtuk a műveleteket (pl. adja őket össze), és az eredményt olvassa be a tárolóba. Más utasítások meg tudtak vizsgálni egy számot, és bizonyos feltételek mellett osztályozni tudták őket aszerint, hogy pozitív vagy negatív. Ha a bemeneti kártyára más programot lyukasztottak, akkor az analitikai motor képes volt már eddigiektől különböző számításokat elvégezni, ami nem volt igaz a differenciálmotor esetében.

Amióta az analitikai motor egy egyszerű assembly nyelven programozható volt, szoftverre volt szüksége. Hogy létrehozza ezt a szoftvert, Babbage egy Ada Augusta Lovelace nevű fiatal nőt fogadott fel, aki a híres brit költő, Lord Byron lánya volt. Így vált Ada Lovelace a világ első programozójává. Az Ada® nevű modern programozási nyelvet az ő tiszteletére nevezték el.

Sajnos, sok tervezőhöz hasonlóan Babbage sem tudta a hardverhibákat teljesen kiküszöbölni. A probléma az volt, hogy neki több ezer nagy precizitással elkészített apró alkatrészre, fogaskerekekre volt szüksége, amit a 19. századi technológia képtelen volt kivitelezni. Mindazonáltal elképzelése korán meghaladta, és még ma is a legtöbb modern számítógép felépítése nagyon hasonlít az analitikai motorhoz, szóval úgy igazságos, ha azt mondjuk, hogy Babbage volt a modern digitális számítógépek (nagy)apja.

A következő nagy fejlődés az 1930-as évek végén történt, amikor egy Konrad Zuse nevű német mérnökhallgató egy sorozat automata számítógépet épített, amelyek elektromágneses reléket használtak. Miután a háború elkezdődött, nem tudott kormányzati támogatást kérni, mivel a kormány bürokratái arra számítottak, hogy megnyerjék a háborút, amilyen gyorsan csak lehet, így az új gép nem készülhetett el addig, amíg az véget nem ért. Zuse nem tudott Babbage munkájáról és a gépeit 1944-ben Berlin bombázásakor az Allied lerombolta, így a munkája nem gyakorolt hatást a későbbi gépekre. Mindezek ellenére a terület egyik úttörője volt.

Röviddel ezután az Egyesült Államokban is tervezett két ember számítógépeket: John Atanastoff az Iowa Állami Akadémiáról és George Stibbitz a Bell Laboratóriumban. Atanastoff gépe korához képest elképesztően fejlett volt. Kettes számrendszer használt, a memóriákat kondenzátorok működtették, amiket folyamatosan frissítettek, nehogy az adatok elvesszenek, ezt a folyamatot ő a “memória frissítésének” nevezte. A modern dinamikus memóriachipek (RAM) tejesen hasonló módon működnek. Sajnos ez a gép soha nem vált működővé. Ily módon Atanastoff Babbage-hez hasonlóan egy látnok volt, aki végül is kudarcot vallott, korának nem megfelelő hardver-technikája miatt.

16.old.

Stibbitz számítógépe, bár sokkal egyszerűbb volt, mint Atanastoffé, valóban működött. Stibbitz 1940-ben a Darmouth Akadémián egy konferencián nyilvános előadást tartott róla. A hallgatóság egyik tagja John Mauchley, a Pennsylvaniai

Egyetem egy ismeretlen fizikaprofesszora volt. Mauchley professzorról a számítástechnika világa később még sokat hallott.

Míg Zuse Stibbitz és Atanastoff automatikus számítógépeket terveztek, egy Howard Aiken nevű fiatalember a Harvardon Ph.D. kutatásainak részeként kézzel dolgozott ki unalmas numerikus számításokat. Miután diplomáját megszerezte, Aiken felismerte annak fontosságát, hogy géppel is képes legyen számításokat végezni. Elment a könyvtárba, felfedezte Babbage munkáját, és elhatározta, hogy relékből építi fel azt az általános rendeltetésű számítógépet, amit Babbage nem tudott megépíteni fogaskerekekből.

Aiken első gépét a Mark I-t a Harvardon készítette el 1944-ben. Tárolója 72 db 23 tízes számrendszerbeli szót tudott tárolni, és egy műveletet 6 másodperc alatt végzett el. Bemeneti és kimeneti egységként lyukszalagot használt. Idővel Aiken elkészítette ennek utódját, a Mark II-t, a relét használó számítógépek elavultak. Elkezdődött az elektronikus számítógépek kora.

1.2.2 Az első generáció - Elektroncsövek (1945-1955)

Az elektronikus számítógépek készítését a II. világháború ösztönözte. A háború korai szakaszában a német tengeralattjárók angol hajók elpusztításával álltak bosszút. A parancsokat Berlinből a német tengernagyok rádión keresztül küldték a tengeralattjáróknak, amikről a britek tudomást szereztek és le is hallgattak. A baj az volt, hogy ezeket az üzeneteket egy ENIGMA nevű gépezet segítségével kódolták, aminek elődjét egy amatőr feltaláló és az Egyesült Államok egy korábbi elnöke, Thomas Jefferson tervezte.

A háború elején a brit hírszerzésnek sikerült szereznie egy ENIGMA gépet a Lengyel Hírszerzéstől, akik a németektől lopták el. Azonban ahhoz, hogy feltörjenek egy üzenetet, hatalmas mennyiségű számításokra volt szükség, mégpedig nagyon kevéssel azután, hogy az üzeneteket befogták, hogy valami értelme is legyen. Hogy megfejtse ezeket az üzeneteket, az angol kormány egy titkos laboratóriumot állított fel, amelyben megépítettek egy COLOSSUS nevű elektronikus számítógépet. A gép tervezésében a híres brit matematikus, Alan Turing nyújtott segítséget. A COLOSSUS 1943-ban vált működőképessé, de miután a brit kormányzat a kutatás gyakorlatilag minden egyes lépését 30 évre katonai titokká minősítette, a COLOSSUS-ág alapvetően zsákutca volt. Csak azért érdemes megjegyezni, mert ez volt a világ első elektronikus, digitális számítógépe.

Zuse számítógépének lerombolásán és a COLOSSUS építésének ösztönzésén kívül a háború az Egyesült Államok számítástechnikájára is hatással volt. A hadseregnek lőtáblázatra volt szüksége a nehéztüzérsége számára. Ezeket a táblázatokat úgy állították elő, hogy nők ezreit bérelték fel, hogy kézi számológép segítségével határozzák meg őket (úgy gondolták, hogy a nők sokkal alaposabbak, mint a férfiak). Mindazonáltal a folyamat időigényes volt, és gyakran fordultak elő benne hibák.

John Mauchleynak, aki jól ismerte Atanastoff és Stibbitz munkáját is, tudomása volt arról , hogy a hadsereg érdeklődik a mechanikus számítógépek iránt. Sok utána következő számítástechnikai tudóshoz hasonlóan, egy javaslatot állított össze, megkérve ezzel a hadsereget, hogy finanszírozzák egy elektronikus számítógép megépítését.

17.old.

A javaslatot 1943-ban elfogadták, és Mauchley és az ő diplomázó tanulója, J. Presper Eckert hozzáfogtak, hogy megépítsenek egy elektronikus számítógépet, amit ők

ENIAC-nak (**Elektronic Numerical Integrator And Computer**) neveztek el. Ez 18000 elektroncsőből és 1500 reléből állt. Az ENIAC 30 tonnát nyomott, és 140 kW volt a teljesítménye. Szerkezetileg, a gépnek 20 regisztere volt, mindegyik egyenként 10 tízes számrendszerbeli szót tartalmazott. (A decimális regiszter nagyon kicsi memória, amely egytől néhány tízes számrendszerbeli maximum számot képes tárolni, kicsit hasonlóan, mint a kilométer-számláló, amely azt követi nyomon, egy autó életében milyen hosszú utat tett meg.) Az ENIAC 6000 többállású kapcsoló létrehozásával és nagyszámú konnektor és valódi erdőnyi kapcsolókábel összekapcsolásával programozták.

A gép nem készült el 1946-ig, amikor is már túl késő volt eredeti rendeltetésére való használatához. Mindezek ellenére, miután a háború véget ért, Mauchley és Eckert engedélyt kaptak arra, hogy nyári egyetemet szervezzenek, ahol bemutatták munkájukat tudományos munkatársainak. Ez a nyári egyetem jelentette a nagy digitális számítógép építése iránti érdeklődés robbanásszerű növekedésének kezdetét.

Ezután a történelmi nyári egyetem után sok más tudományos kutató kezdett elektronikus számítógépek építésébe. Az első működőképes az EDSAC (1949) volt, amelyet Maurice Wilkes épített a Cambridgei Egyetemen. A többiek közé sorolható a JOHNIAC a Rand Részvénytársaságtól, az ILLIAC az Illinosisi Egyetemen, a MANIAC a Los Alamosi Laboratóriumban és a WEIZAC a Weizmann Intézetben Izraelben.

Eckert és Mauchley hamarosan az utódon, az **EDVAC**-on (**Electronic Discrete Variable Automatic Computer**) kezdtek el dolgozni. Bár a tervek súlyosan károsodtak, mikor ők elhagyták a Pennsylvaniai Egyetemet, hogy létrehozzanak egy gyorsan fejlődő vállalatot, az Eckert-Mauchley Számítógépes Társulatot Philadelphiában (a Szilíciumvölgyet még nem fedezték fel). Egy sor fúzió után ez a vállalat vált a modern Unisys Társulattá.

Törvényes részletként, Eckert és Mauchley benyújtották a szabadalmazásra vonatkozó kérelmet, állítva, hogy ők találták fel a digitális számítógépet. Visszatekintve, nem lett volna rossz egy ilyen szabadalmat birtokolni. A pereskedés évei után a bíróság úgy határozott, hogy az Eckert-Mauchley szabadalmi kérvény érvénytelen, és hogy John Atanastoff találta fel a digitális számítógépet, még ha nem is szabadalmaztatta.

Míg Eckert és Mauchley az EDVAC-ot építették, az ENIAC tervein dolgozó emberek egyike, John Von Neumann, a Magasabb Tanulmányok Princetoni intézetébe ment, hogy megépítse az EDVAC egyéni verzióját, az IAS gépet. Neumann ugyanolyan nagy zseni volt, mint Leonardo Da Vinci. Sok nyelven beszélt, kiváló fizikus és matematikus volt, és mindenre pontosan vissza tudott emlékezni, amit valaha hallott látott vagy olvasott. Képes volt szó szerint idézni olyan könyvek szövegét, amit évekkel ezelőtt olvasott. Amikor elkezdett érdeklődni a számítástechnika iránt, már a világ legkimagaslóbb matematikusa volt.

A dolgok közül elsőként az vált nyilvánvalóvá számára, hogy a számítógépet nagyszámú kapcsolóval és kábelekkel programozni unalmas és merev.

***[18-21]

a számítógép memóriája az adatokkal együtt. Azt is észrevette, hogy az ENIAC által használt ügyetlen decimális aritmetika, melyben mindegyik számjegy 10 vákuumcsővel van ábrázolva, helyettesíthető párhuzamos bináris aritmetika használatával.

Az alapgondolat, melyet legelőször felvázolt, ma Neumann gépként ismert. Ez az EDSAC-ban, az első programtárolású számítógépben volt használva, és most is, több mint egy fél évszázaddal később, még mindig az alapja majdnem az összes digitális számítógépnek. Ennek a konstrukciónak és az IAS gépnek, amit Herman Goldstine-nal együttműködve építettek olyan óriási hatása volt, hogy megéri röviden leírni. Az architektúra vázlata az 1.-5. ábrán látható.

A Neumann gépnek öt elemi része volt: a memória, az aritmetikai-logikai egység, a vezérlő egység, a bemeneti és a kimeneti berendezések. A memória 4096 szóból állt, szavanként 40 bitet tárolva, melyek 0-k vagy 1-k voltak. Mindegyik szó vagy két 20 bites utasítást vagy egy 40 bites előjeles egész számot tartalmazott. Az utasításokban 8 bit az utasítás típusának meghatározására és 12 bit a 4096 memóriaszó egyikének meghatározására volt szánva.

Az aritmetikai-logikai egységen belül volt egy speciális 40 bites regiszter, az akkumulátor. Egy tipikus utasítás egy memóriaszót tett az akkumulátorba vagy az akkumulátor tartalmát tárolta a memóriában. Ez a gép nem tudott lebegőpontos aritmetikát végezni, mert Neumann úgy gondolta, hogy egy kompetens matematikusnak képesnek kell lennie a tizedespontot (vagyis a binárispontot) nyomon követnie a fejében.

Körülbelül ugyanabban az időben, amikor Neumann megépítette az IAS gépet, az MIT kutatói szintén alkottak egy számítógépet. Az IAS-szel ellentétben az MIT gép, a Whirlwind I. 16 bites szóval rendelkezett és valós idejű vezérlésre tervezték. Ez a projekt vezetett a Jay Forrester által feltalált mágnesmagú memóriához és tulajdonképpen az első kereskedelmi kisszámítógépekhez.

Míg ezek történtek, az IBM egy olyan kis cég volt, amelyik lyukkártya lyukasztók és mechanikus kártyaválogató gépek gyártásával kezdett el foglalkozni. Habár az IBM adta az Aiken pénzének egy részét, nem volt túlságosan érdekelve a számítógépekben, amíg el nem készítette a 701-et 1953-ban, jóval az után, hogy Eckert és Mauchley cége első volt a kereskedelmi piacon az UNIVAC számítógépével. A 701 számítógép 2048 darab 36 bites szóval rendelkezett, két utasítással szavanként. A tudományos gépek sorozatában ez volt az első, ami egy évtizeden belül dominálta az ipart. Három évvel később jött a 704, amelyiknek 4K magmemóriája volt, 36 bites utasításai és lebegőpontos hardvere. 1958-ban az IBM megkezdte az utolsó vákuumcsöves gépének a gyártását, a 709-ét, ami alapvetően egy feljavított 704 volt.

1.2.3. A második generáció – Tranzisztorok (1955-1965)

A tranzisztort John Bardeen, Walter Brattain és William Shockley találták fel 1948-ban a Bell Laboratóriumokban, amiért 1956-ban fizikai Nobel-díjat kaptak. Tíz éven belül a tranzisztor forradalmasította a számítógépeket, és az 1950-es évek végére a vákuumcsöves számítógépek elavultnak számítottak. Az első tranzisztoros számítógépet az MIT Lincoln Laboratóriumában építették, egy 16 bites gép volt a Whirlwind I. nyomában. TX-0-nak hívták és csupán egy eszköznek szánták, hogy teszteljék a sokkal különlegesebb TX-2-t.

A TX-2 sose vitte sokra, de a laboratóriumban dolgozó mérnökök egyike, Kenneth Olsen alapított egy céget, a Digital Equipment Corporation-t (DEC) 1957-ben, hogy egy TX-0-hoz hasonló gépet készítsen. Ez négy évvel azelőtt volt, hogy ez a gép, a PDP-1 megjelent, főleg azért, mert a merész kapitalisták, akik a DEC-t alapították mereven hittek abban, hogy a számítógépek számára nincs piac. Ehelyett a DEC főleg kis áramkörlapokat árult.

Amikor a PDP-1 végre megjelent 1961-ben, 4K-s 18 bites szavai voltak és a ciklusidő 5 µsec volt. Ez a teljesítmény a fele volt az IBM 7090-ének, a 709-es tranzisztoros utódja teljesítményének, ami ebben az időben a leggyorsabb számítógép volt a világon. A PDP-1 120 ezer dollárba került, a 7090 milliókba.

A DEC PDP-1-esek tucatjait adta el, és a kisszámítógép-ipar megszületett. Az első PDP-1-esek egyikét a MIT-nek adták el, ahol gyorsan megragadta néhány fiatal bimbózó tehetség figyelmét, akik a MIT-en olyan gyakoriak. A PDP-1-esek sok újítása közül az egyik a vizuális kijelző és a pontok ábrázolásának a képessége bárhol az 512-szer 512-es képernyőjén. Hamarosan a hallgatók programozták a PDP-1-et, hogy űrháborút játsszanak, és a világnak megvolt az első videojátéka.

Pár évvel később a DEC bemutatta a PDP-8-at, ami egy 12 bites gép volt, de sokkal olcsóbb, mint a PDP-1 (16 ezer dollár). A PDP-8-nak egy óriási újítása volt: az egyszerű busz, az omnibusz, ahogy az 1.-6.-os ábrán is látható. A busz párhuzamos kábelek gyűjteménye, amelyeket a számítógép részeinek összekötésére használnak. Ez az architektúra egy óriási eltávolodás volt a memóriaközpontú IAS géptől és ezt azóta átvette majdnem minden kisszámítógép. A DEC azzal, hogy 50000 PDP-8-ast adott el, tulajdonképpen megalapozta vezető szerepét a kisszámítógép-iparban.

Eközben az IBM reakciója a tranzisztorra a 709-es tranzisztoros verziójának, a 7090-nek a megépítése volt, mint már fentebb is megemlítettük, majd később a 7094-esé. A 7094-nek 2 µsec-os ciklusideje és 32K-s, 32 bites szavakból álló magmemóriája volt. A 7090 és a 7094 jelentették az ENIAC típusú gépek végét, de a tudományos számításokban még az 1960-as években is domináltak.

Ezzel egy időben, hogy az IBM egy fontosabb hatalom lett a tudományos számításokban a 7094-gyel, óriási összegeket keresett egy kis üzlet-orientált gép, az 1401-es eladásával. Ez a gép mágnesszalagokat tudott írni és olvasni, kártyákat lyukasztani és olvasni, és majdnem olyan outputot tudott nyomtatni, mint a 7094, és az ára csak töredéke volt. Tudományos számításokra szörnyű volt, de üzleti feljegyzések tárolására tökéletes.

Az 1401-es szokatlan volt abban, hogy nem voltak regiszterei vagy akár fix szóhosszúsága. A memóriája 4K-s 8 bit bájtos volt. Mindegyik bájt egy 6 bites karaktert tartalmazott, egy adminisztratív bitet és egy bitet, hogy jelezze a szó végét. Egy MOVE utasításnak pl. volt egy forrás és egy cél címe, amíg elért egy 1-re állított szó vége bitet.

1964-ben egy új induló cég, a CDC bemutatta a 6600-at, egy gépet, amelyik majdnem egy nagyságrenddel gyorsabb volt, mint a hatalmas 7094. A "számaprítóknak" szerelem volt ez első látásra és a CDC elindult a sikere útján. A gyorsaságának a titka, és az ok, amiért sokkal gyorsabb volt a 7094-esnél az volt, hogy a belső CPU egy magasan párhuzamos gép volt. Több funkciós egysége is volt összeadás elvégzésére, mások szorzásra, mások osztásra és ezek közül mindegyik párhuzamosan futhatott. Habár pontos programozást igényelt az, hogy a legtöbbet hozzák ki belőle, de valamennyi munkával lehetővé vált, hogy akár 10 utasítást végezzen egyszerre.

Mintha ez nem lett volna elég, a 6600-ba számos kis számítógép volt beépítve,

hogy segítse, mint pl. Hófehérke és a hét törpe. Ez azt jelentette, hogy a CPU az összes idejét számok feldarabolásával töltheti, a munka végrehajtását, és az input/output műveletek nagy részét a kis számítógépekre hagyva. Visszatekintve a 6600-as évtizedekkel megelőzte saját korát. A modern számítógépekben található kulcsötletek nagy része visszavezethető egyenesen a 6600-asra.

A 6600 tervezője, Sejmour Cray egy legendás alak volt, egy csoportban Neumannal. Egész életét egyre gyorsabb számítógépek építésére szentelte, amiket ma szuperszámítógépeknek hívunk, beleértve a 6600-at, a 7600-at és Cray-1-et. Ő találta fel a híres autóvásárlási algoritmust: a lakhelyedhez legközelebbi eladóhoz elmész, az ajtóhoz legközelebbi autóra rámutatsz, és azt mondod: "Azt megveszem". Ez az algoritmus a legkevesebb időt pazarolja nem fontos dolgokra (pl. autóvásárlás) és a legtöbb időt fontos dolgok elvégzésére (mint a szuperszámítógépek tervezése).

Több más számítógép volt ebben a korban, de az egyik kiemelkedik egy különleges ok miatt és ezt érdemes megemlíteni: a Burroughs B5000. Az olyan gépek tervezői, mint a PDP-1, 7094 és a 6600 el voltak foglalva a hardverrel, hogy vagy olcsóbbá tegyék (DEC), vagy gyorsabbá (IBM, CDC). A szoftver majdnem teljesen lényegtelen volt. A B5000 tervezői más irányt választottak. Építettek egy gépet specifikusan azzal a szándékkal, hogy Algol 60-ban (a Pascal előfutára) lehessen programozni, és több sajátosságot helyeztek el a hardverben, hogy könnyítsék a fordító feladatát. Az ötlet, hogy a szoftver is számít, megszületett. Sajnos ezt majdnem egyből el is felejtették.

1.2.4. A harmadik generáció – Integrált áramkörök (1965-1980)

A szilikon integrált áramkörök feltalálásával, amit Robert Noyce talált fel 1958-ban, több tucat tranzisztor elhelyezése vált lehetővé egyetlen chipen. Ez a csomagolás lehetővé tette, hogy olyan számítógépeket építsenek, amelyek kisebbek, gyorsabbak és olcsóbbak, mint tranzisztoros elődjeik.

1964-re az IBM lett a vezető számítógépes cég és egy nagy problémája volt két nagyon sikeres gépével, a 7094-gyel és az 1401-gyel: annyira inkompatibilisek voltak, amennyire két gép lehet. Egyik egy nagysebességű számdaraboló volt párhuzamos bináris aritmetikát használva 36 biten és a másik egy dicsőített input/output processzor volt soros decimális aritmetikát használva változó hosszúságú szavakon a memóriában. A testületi vásárlóik közül soknak mind a kettő meg volt és nem tetszett nekik az, hogy két külön programozó osztályt kell működtetniük, amikben semmi közös nincs.

Mikor elérkezett az idő e két sorozat lecserélésére az IBM egy radikális lépést tett. Egyetlen termékvonalat vezetett be, a System/360-at, ami az integrált áramkörökön alapult, ami mind tudományos, mind kereskedelmi számításokra volt tervezve. A System/360 sok újítást tartalmazott, ezek közül a legfontosabb, hogy ez egy kb. fél tucat gépből álló, ugyanazt az assembly nyelvet használó gépek családja volt növekvő mérettel, és teljesítménnyel. Egy cég ki tudta cserélni az 1401-ét egy 360 Model 20-ra és a 7094-ét egy 360 Model 75-re. A Model 75 nagyobb és gyorsabb volt (és drágább is), de az egyiken írott szoftver, elvileg, futott a másikon is. Gyakorlatban, egy kis modellen írt szoftver elfutott a nagy modellen, de ha egy kisebb modellre tették át, a program nem fért bele a memóriába. Így is ez egy nagy előrelépés volt a 7094 és a 1401 szituációjához képest.

***[22-25]

A kezdő 360-as család néhány jellemzőjét mutatja be az 1-7. ábra. Egyéb modelleket később mutatunk be.

TULAJDONSÁG	**Modell 30**	**Modell 40**	**Modell 50**	**Modell 65**
Relatív teljesítmény	1	3.5	10	21
Forgási idő	1000	625	500	250
Maximális memória(KB)	64	256	256	512
Byte áramlás/ciklus	1	2	4	16
Adatcsatornák max. száma	3	3	4	6

1-7. ábra Az IBM 360-as termékvonal kezdeti ajánlata.

A 360-as másik fő újítása a **multiprogramozás** volt, ami számos programot tartalmazott egyszerre a memóriában, tehát amíg valaki egy input/output-ra várt, hogy befejeződjön, addig másik programmal is tudott dolgozni.

A 360-as volt az első olyan gép, ami versenyképes volt a többi számítógéppel. A kisebb gépek versenyképesek voltak az 1401-sel, míg a nagyobbak a 7094-sel, így a vásárlók folyamatosan tudták futtatni az elöregedett programjaikat a gépeken, mert az átkonvertálta a 360-asra. Néhány modell gyorsabban futtatta az 1401-es programjait, mint maga az 1401-es, így sok vásárló soha nem is újította meg a programjait. A vesenyképesség fenntartása azért volt könnyű a 360-nál, mert mind a kezdeti modellek, mind a későbbiek nagy része mikroprogramozású volt. Ennek érdekében az IBM-nek 3 utasításcsomagot kellett írnia: egyet az eredeti 360-asra, egyet az 1401-esre és egyet a 7094-esre. Ez a rugalmasság volt az egyik fő oka, hogy a mikroprogramozást bevezették.

A 360-as megoldotta a 2-es és a 10-es számrendszer közötti ellentmondást egy kompromisszummal: a gépnek 16 db 32 byte-os regisztere van a bináris számrendszerhez, de a memóriája byte orientált, úgy mint az 1401-esé. Ennek szintén 1401-es stílusú utasítás sorozata van a különböző méretű mozgatható recordoknak a memória körül.

Abban az időben a másik fő tulajdonsága volt a 360-asnak a hatalmas cím kiterjesztés, ami 2^{24} byte (16Mbyte) volt .Akkoriban a memória byte-ként nagyon sokba került, a 16 Mbyte pedig elérhetetlennek látszott. Sajnos a 370-es, 4300-as, 3080-as és a 3090-es szériák, amelyek a 360-ast követték, ugyanazt a szerkezeti felépítést használták. A 1980-as évek közepére a 16 Mbyte-os határ élő problémává vált, és az IBM-nél a kompatibilitás rovására ment a 32 bites címjegyzékhez szükséges új 2^{32} byte memória kialakítása. Utólag bebizonyosodott, hogy, míg 32 bit kiterjesztsű szavaik és regisztereik vannak, addig valószínüleg 32 bites címeik lehetnek, de abban az időben senki nem tudott elképzelni 16 Mbyte-os gépet. Az előrelátás hiba volt az IBM részéről, úgy mintha egy modern személyi számítógép eladónak csak 32 bites címkiterjesztése lenne. Néhány éven belül a PC-knek több mint 4GB-os memóriával kell rendelkezniük, amikor a 32 bit címkiterjesztés tűrhetetlenül kicsivé válik.

A minikomputer világ is hatalmas lépést tett előre a 3. generációban a PDP-11-es széria DEC bevezetésével, és a 16 bites PDP-8 utóddal. Több esetben is a PDP-11-es széria mintha csak a kistestvére lenne a 360-asnak, csakúgy mint a PDP-1-es a 7094-esnek. A 360-as és a PDP-11-es is szövegorientált regiszterű és byte orientált memóriájú, mindkettőnek tekintélyes az ár/teljesítmény aránya. A PDP-11 rendkívül sikeres volt, különösen az egyetemeken, és biztosította a DEC vezető helyét a többi minikomputer gyártóknál.

1.2.5 A 4. generáció-VLSI(Nagyon Széles Skálájú Integráció (1980-?))

Az 1980-as évekre a VLSI lehetővé tette, hogy az ezrekből az első 10 majd 100 tranzisztort később milliókat tegyenek egy egyszerű chipre. Ez a fejlődés kisebb és gyorsabb komputerekhez vezetett. A PDP-1-es előtt a számítógépek olyan nagyok és drágák voltak, hogy a cégeknek és az egyetemeknek egy külön speciális részleget kellett fenntartani, amiket **számítógép központoknak** neveztek, hogy tudják őket működtetni. A minikomputer megjelenésével már egyes részlegek is képesek voltak sajaát számítógépet venni. Az 1980-as évekre az árak annyira leestek, hogy egyes embereknek is lehetett már saját gépük. A PC korszak ezzel elkezdődött.

A PC-k használata nagyon eltér a nagy számítógépekétől. Ezeket szövegszerkesztésre, lapterjesztésre és számos magas szintű alkalmazásra használták, amire a nagy gépek nem túl jók.

Az első PC-ket általában egységcsomagonként árusították. Minden csomag tartalmazott egy nyomtatott áramkötáblát, egy köteg chippet, többnyire magában foglalt egy Intel 8080-ast, néhány kábelt, erősítőt és esetleg egy 8 inches floppy lemezt. A részek összeszerelése. a gép összeállítása a vásárlóra maradt. Software-rel nem volt ellátva.Ha valaki akart volna, annak magának kellett megírnia. Később a Gary Kildall által írt CP/M operációs rendszer nagyon népszerű lett a 8080-ason. Ez egy igazi (floppy) lemezes operációs rendszer volt, file szervezésű, és a vezényszavakat közvetlenül a klaviatúráról írhatta be a felhasználó.

Másik korai PC-k voltak az Apple és később az AppleII, amiket Steve Jobs és Steve Wozniak terveztek egy híressé vált garázsban. Ez a gép rendkívül népszerű volt az otthoni felhasználók körében, és komoly "játékossá" lett csaknem éjszakákon át.

Hosszas tanácskozások és megfigyelések után, amit más cégek tettek, az IBM, mint a számítógép ipar domináns ereje, végül elhatározta, hogy kiveszi a részét a PC üzletből. Mivel a gép teljes tervezését a semmiből kellett kezdeni csak IBM alkatrészek felhasználásával, amelyek elavultak voltak, az IBM rá nnem jellemző lépést tett.Megbízták az IBM egyik vezetőjét Philip Estridge-t, hogy egy zsák pénzzel vonuljon el a bürokrácia látóköréből a szövetséges Armonk-i (NY) főhadiszállásra, és addig ne jöjjön vissza, amíg szert nem tesz egy működő PC-re. Estridge felépített egy üzletet a Főhadiszállástól messze, Boca Ratonban (FL), kiválasztotta az Intel 8080-ast, mint a saját CPU-ját, és épített egy IBM PC-t a kereskedelmi forgalomban lévő alkatrészekből. A gépet 1981-ben mutatták be, és hamarosan a történelem egyik legkeresettebb számítógépévé vált.

Az IBM tett még egy rá nem jellemző dolgot, amit később meg is bánt. Ahelyett, hogy a gép terveit teljes titokban tartották volna (vagy legalább szabadalommal levédték volna), mint ahogy az szokás, megjelentették a komplett terveket a kapcsolási rajzokkal együtt egy 49 $-os könyvben. Az ötlet az volt, hogy lehetővé tegyék más számítógépes cégeknek az IBM PC-hez való csatlakozást, ezzel is

növelve annak népszerűségét és rugalmasságát. Az IBM szerencsétlenségére mivel a tervek most már teljesen publikusak voltak, és valamennyi része könnyen beszerezhető volt a kereskedelmi forgalomban, számos más cég elkezdte a PC-k **klónjait** gyártani, gyakran sokkal olcsóbban, mint az IBM. Így indult el a sorozatgyártás.

Bár az egyébb cégek nem Intel CPU-t alkalmaztak a PC-kben, beleértve a Commodore-t, Apple-t, Amigra-t és Altari-t, az IBM súlya olyan nagy volt, hogy ezeket a cégeket az IBM lesöpörte a piacról. Csak néhányan maradtak meg a piacon mint pl: a mérnöki állomások vagy a szuperkomputerek.

Az IBM PC első verziói MS-DOS operációs rendszerrel voltak felszerelve, amelyeket a (then-tiny) Microsoft Corporation működtetett. Mint ahogy az Intel képes volt egyre erősebb CPU-kat előállítani, az IBM és a Microsoft ki tudta fejleszteni az MS-DOS jogutódját, amit OS/2-nek neveztek, amely képes volt grafikus feldolgozásokra csakúgy mint az Apple Macintoshé. Miközben a Microsoft is kifejlesztette a saját Windows operációs rendszerét és ez az MS-DOS csúcsára futott, addig az OS/2 nem volt sikeres. Egy szónak is száz a vége az OS/2 nem sikerült, az IBM és a Microsoft nyilvánosan összevesztek, és a Windows Microsoft fejlesztésként vált sikeressé. Ahogy az apró Intel és a még apróbb microsoft elérte az IBM trónfosztását, ami a világtörténelem egyik legnagyobb és legerőteljesebb cége volt, kétségtelenül példaértékű, ami részletesen érdemes elmondani a manager iskolákban az egész világon.

Az 1980-as évek közepére egy RISC nevű új fejlesztés kezdi átvenni a komplikált felépítésű CISC helyét, amely jóval egyszerűbb (de gyorsabb). Az 1990-es években a szuperskálájú CPU-k kezdenek megjelenni. Ezek a gépek egyidőben képesek összetett feladatokat végrehajtani, gyakran olyan parancsokat is, amelyek eltérőek a programjukban leírtaktól. A második fejezetben bemutatjuk a CISC és a RISC és a szuperskála alapelveit, és a könyv folyamán megtárgyaljuk a részleteket.

1.3 KOMPUTER PARK

Az előző részben rövid áttekintést adtunk a számítógép történetéről. Ebben a részben a jelenről lesz szó és kitekintünk a jövőre is. Bár a PC-k a legismertebb komputerek, napjainkban léteznek olyan gépek is amelyekről nagyon nehéz röviden elmondani működésüket és felhasználásukat.

25.oldal

1.3.1 Technikai és gazdasági erők

A számítógép gyártás soha nem látott léptekkel fejlődik. Az elsődleges mozgatóereje az, hogy a chip gyártók képesek egyre több tranzisztort elhelyezni egy chipre évről-évre. Több tranzisztor, amelyek pici elektronikus kapcsolók, nagyobb memóriát és nagyobb erejű processzort jelentenek.

A technológiai haladás mértéke a Moore-törvénynek nevezett megfigyelés alapján modellezhető, amely a nevét Gordon Moore-ról, az Intel egyik alapítójáról és vezetőjéről kapta, és amit 1965-ben fedezett fel. Miközben Moore a beszédeit állította össze az ipari társaságnak, felismerte, hogy a memória chippek új generációját 3 évenként mutatják be. mivel minden új generációban 4-szer annyi memória van, mint az elődjében, rájött, hogy az 1 chipre tehető tranzisztorok száma egy állandó szerint

növekszik, és megjósolta, hogy ez a növekedés még évtizedekig fog tartani.Napjainkban Moore-törvényét gyakran úgy értelmezik, mint a tranzisztorok számának 18 havonkénti megduplázódása. Megjegyezzük, hogy ez egyenlő a tranzisztorok számának évenkénti kb. 60%-os növekedésével. A memória chippek mérete, és a bemutatásuk dátuma az 1.8 ábrán látható, amely igazolja, hogy a Moore-törvény még mindig működik.

1.8 ábra A Moore- törvény az 1 chipre tehető tranzisztorok számának évenkénti 60%-os növekedését jósolta meg. Az adatpontok, amelyeket az ábrán felhasználtunk az 1 bitben található memória helyeket mutatják.

Természetesen a Moore-törvény nem egy általános érvényű törvény, csupán egy tapasztalati megfigyelés arról, hogy a stabil tudományú fizikusok és a folyamat mérnökök milyen előrehaladást tudnak elérni, és annak a jóslata, hogy ezt az előrehaladást ugyanilyen tempóban tudják biztosítani a jövőben is. Sok megfigyelő azt várja el a Moore-törvénytől, hogy még a XXI. században is jól működik majd, talán 2020-ig. Ennél a pontnál a tranzisztorok már csak néhány megbízható atomból fognak állni, jóllehet a kvantum komputerizálás előnyei lehetővé teszik, hogy 1 bit tárolására egy magányos elektron is elegendő legyen.

A Moore-törvény olyasvalamit adott, amit néhány közgazdász **erkölcsös körforgásnak** nevez. A technológiából adódó előnyök (tranzisztorok száma/chip) jobb termékekhez és alacsonyabb árakhoz vezettek. Az alacsonyabb árak új felhasználásokat eredményeztek (senki sem készített video játékokat a számítógépekhez, amíg azok 10 millió dollárba kerültek). Az új felhasználások új piacokat nyitottak és új cégek ismerték fel a számítógépek előnyeit.

***[26-29]
Fordítás, 26-29

Az ilyen vállalatok léte versengéshez vezet, ami olyan gazdasági igényt támaszt a jobb technológiák iránt, amivel legyőzhetik a többieket. Így a kör bezárul.

Egy újabb vezető technológiai előrelépés Nathan első software törvénye (Nathan Myhrvold, a Microsoft ügyvezetője). Eszerint: “A Software gáz. Kitágul, hogy kitöltse a tárolőedényét”. Az 1980-as évekre visszamenőleg a szövegszerkesztést olyan programokkal csinálták, mint a troff (amit a könyvhöz is használtak). A troff több tíz KByte memóriát foglal el. A modern szövegszerkesztők több tíz Mbyte-ot birtokolnak. A jövőbeliek kétségkívül több tíz Gbyte igényűek lesznek. A software ami egyre inkább olyan sajátságokkal rendelkezik (hasonlóan a hajók oldalára rakódó kagylóhoz) egy állandó igényt jelent a gyorsabb proceszorokra, a nagyobb memóriára és több I/O kapacitásra.

Amíg a csipenkénti tranzisztorból származő nyereség drámainak mondtható az elmúlt években, az egyéb kompjuter technológiákból származó profit nőtt. Például az IBM PC/XT-t 1932-ben 10 MB-s hard disc-kel vezették be. Törvényszerűen a jelenlegi desktop rendszerek ennél valamivel nagyobbak. A lemezek fejlesztésének mérése már nehezebb, hiszen számos paraméter van (kapacitás, adatok aránya, ár, stb.) de szinte minden mérési eredmény azt mutatná, hogy a kapacitás azóta legalább 50%-kal nőtt évente.

Egy másik terület, mely látványos nyereséget termel a mai napig a telekommunikáció és a hálózat. Kevesebb mint 20 év alatt 300 bit/perc modemektől eljutottunk az analóg 56 kbps modemekig, ISCN telefonvonalak 2x64 kbps-nál, a száloptikás hálozatok már jóval 1 Gbyte/perc fölett vannak. Az Atlanti-óceánon áthúzódó száloptikás telefonkábelek, mint a TAT-12/13 , kb 700 millió dollárba kerülnek, 10 éveig tartanak 300,000 egyidejű hívást képesek továbbítani, ami kevesebb, mint 1 centet jelent egy 10 perces interkontinentális hívásnál. Laboratoriumi kisérletek szerint bizonyíthatóan megvalósítható a száloptikás 1 terabit/perc (10^{12} bit/perc)-os komunikációs rendszerek amelyek 100 km-t is meghaladhatnak erősítő használata nélkül. Az Internet exponenciális növekedését nem szükséges kommentálnunk.

1.3.2 A számítógép válszték

Richard Hamming, a Bell Laboratórium egy régebbi kutatója, egyszer megfigyelte, hogy a mennyiség nagyságrenddel való változtatása minőségi változást okoz. Ekkép egy versenyautó, amely 1000 km/óra sebességgel képes menni a Nevada sivatagban, alapvetően különböző fajtáju gép, mint egy normál autó, amely 100 km/óra sebességgel halad az autópályán. Hasonlóképpen egy 100 emeletes felhőkarcoló nem csak egy arányosan felnagyított 10 emeletes lakóház. És a számítóképeknél nem 10-szeres szörzókról, hanem a több mint 3 évtized folyamán milliós szorzókról beszélünk.

A nyereség Morre törvénye szerint számos módon felhasználható. Az egyik mód, hogy állandó ár mellett növekvő erősségü számítógépeket építenek. Egy másik megközelítés, hogy ugyanaz a számítógépet minden évben egyre alacsonyabb áron állítják elő. A számítógépipar megtette mindkettőt s még többet is, bizonyítja ezt a ma elérhető számítógépek széles választéka. Egy nagyon “durva” kategorizációját a ma forgalomban léjvő számítógépeknek a következő ábra nyújtja:

Tipus	Ár (dollárban)	Példa az alkalmazásra
Eldobható (egyszer használatos) szg.	1	üdvözlő lapok
Rögzített számítógépek	10	órák, kocsik, készülékek
Számítógépes játékok	100	házi videójátékok
Személyi számítógépek	1000	asztali szg. vagy hordozható szg.
Szerver (kiszolgáló)	10000	hálózati szerver
Irodai rendszer	100000	ágazati miniszuperszg.
Mainframe (óriás számítógép)	1 milló	adathalomfeldolgozás egy bankban
Szuper számítógép	10 millió	hosszútávu időjárás előrejelzés

Táblázat: Jelenleg elérhető számítógépek átfogó csoportosítása, az árakat a viszonyítást tükrözik.

Az alsó végen olyan önálló csipeket találunk az üdvözlőkártyák belsejébe ragasztva, melyek eljátszák a “Boldog Születésnapot”, az “Itt jön a menyasszonyt” vagy néhány egyaránt szörnyő versikét. A szerző nem figyelt még fel részvétnyilvánító kártyákra, melyek gyászzenét játszanak, de most hogy beleültettük a közösségi gondolatba, hamarosan várja ezt. Mindenkinek aki milliódolláros óriásszámítógépekkel nőtt fel, az eldobható számítógépek kétségkívűl jelen vannak (mi van a szemetesládákkal, melyek az aluminium dobozok újrafelhasználására emlékeztetnek?)

Következőként a létrán vannak olyan számítógépeink, melyek telefonok, televíziók és mikrohullámu sütők belsejében vannak elhelyezve, vagy áppen CD lejátszók, játékok, babák és ezer egyébb eszközben. Néhány éven belül minden elektromos készülékben egy számítógép lesz. Az “elrejtett” számítógépek száma milliókra lesz becsülve, eltörpítve minden más számítógépet a nagyságrenddel való kombinálás által. Ezek a számítógépek egy processzorral rendelkeznek, kevesebb mint egy Mbyte memóriával és néhány I/O képességgel, mindez egy önálló kis csipen, melyet néhány dollárért árulnak.

Egy lépéssel feljebb a videójátékok találhatók. Ezek rendes számítóképek speciális grafikai képességgel, de korlátozott software-rel és szinte semmi kiterjeszthetőséggel rendelkeznek. Szintén ezen árkategóriában találhatók meg a személyi szervezők, hordozható digitális aszisztensek és hasonló palmtop számítógépes eszközök, éppúgy mint hálózati számítógépek és web terminálok. Ami ezekben a számítógépekben közös az az, hogy tartalmaznak egy processzort, néhány Mbyte memóriát, egy bizonyos kivetítőt (lehetőleg egy tv készülék) és nem sok egyébb dolgot. Ez teszi őket olyan olcsóvá.

Következőként elérkezünk a személyi számítógépekhez, amire a legtöbb ember gondol, amikor a “számítógép” terminust hallja. Ezek magukba foglalják mind az asztali számítógépeket, mind pedig a notebookokat. Ezek általában több megabájt memóriával készülnek, egy merev lemezzel, mely néhány Gbyte adatot tartalmaz, CD-ROM meghajtóval, modemmel, hangkártyával és egyébb perifériákkal. Gondosan kidolgozott operációs rendszerrel rendelkeznek, sok kiterjesztési lehetőséggel és széles skálában elérhető software-rel. Azokat, amelyekbe Intel CPU van beépítve gyakran “személyi számítógépeknek” hívják, míg a különböző fajtáju CPU-val rendelkezőket “munkaállomásoknak”, de fogalmilag van egy kis különbség a két fajta között.

Felerősített személyi számítógépeket vagy munkaálomásokat gyakran használnak

hálózati szerverként mind helyi hálózatokhoz (tipikusan egy önálló vállalaton belül) és az Internethez. Ezek önálló processzor vagy többszörös processzor formákban készülnek, néhány Gbyte memóriáig, sok Gbyte-os merevlemez hellyel és gyors hálózati kapacitással. Néhányuk több tucat vagy több száz bejövő hívást tud fogadni egyszerre.

A kis multiprocesszor szerverek mellett olyan rendszereket is találunk, melyeket úgy hívnak, hogy **"NOW" (Networks of Workstations)** vagy **"COW" (Clusters of workstations)** rendszerek. Szabványos személyi számítógépekből vagy munkaállomásokból állnak, melyek Gbyte/másodperc hálóztok által vannak összekötve, és olyan speciális softwaret futtatnak, amely az összes gépet engedi együtt dolgozni ugyanazon a problámán gyakran a tudományban vagy a technikában. Könnyedén skálázhatóak egy maréknyi géptől kezdve a több ezerig. Az alacson áruk következtében egyes ágazatok rendelkezhetnek ilyen gépekkel, amelyek ténylegesen miniszuperszámítógépek.

Elérkeztünk az óriásszámítógépekhez, szobányi méretű számítógépek, melyek az 1960-as éveket idézik. Sok esetben ezek a gépek egyenes leszármazottaik az IBM 360-as óriásszámítógépeinek melyet évtizedekkel ezelőtt készítettek. Legnagyobb részben nem sokkal gyorsabbak, mint az óriási szerverek, de általában több I/O kapacitásuk van és el vannak látva hatalmas lemez farmokkal, melyek gyakran tartalmaznak egy Tbyte adatot vagy többet ($1TB=10^{12}$ byte). Habár elképeztően drágák, mégis gyakran alkalmazzák őket, köszönhetően a tömérdeknyi befektetésnek a softwarebe, adatokba, operációs módokba és személyzetbe, amiket képviselnek. Sok vállalat olcsóbnak találja, ha csak időnként fizet náhány millió dollárt egy újért, mint hogy azokat az erőfeszítéseket figyeljék, amelyek a kisebb gépek alkalmazásához szükséges újraprogramozással járnak.

Ez az a számítógépcsoport, amelyik mostan a hirhedt 2000. év problémájához vezetett, melyet a COBOL programozók okoztak az 1960-az és1970-es években azáltal, hogy az évet 2 decimális szájegyként jelentették meg (memória megtakarítás céljából). Soha sem képzelték, hogy a software-eik 3-4 évtizednél tovbb lesznek használva. Sok vállalat követte el ugyanazt a hibát azáltal, hogy egyszerűen 2 újabb számjegyet adtak az évhez. A szerző ezzennel megjövendöli a civilizáció végét ahogyan ezt tudjuk 9999 dec. 31-én éjfélkor, amikor a 8000 éves jól kihasznált COBOL programok egyszerre összeomlanak.

Az óriásszámítógépek után jönnek az igazi szuperszámítógépek. Elképesztően gyors CPU-kkal rendelkeznek, sok Gbyte főmemóriával és nagyon gyors lemezekkel és hálózatokkal. Mostanában a szuperszámítógépek nagy része vált sok párhuzamos géppé, nem túlzottan különbözővé a COW tipustól, de gyorsabb és több komponenssel. A szuperszámítógépeket a tudományban és technikában fellépő számításos intenzív problémák megoldására használják, mint pl. összeütköző galaxisok szimulálására, új gyógyszerek szintetizálázára vagy a levegő egy repülőgép szárnya körüli áramlásának modellezésére.

1.4. Példa számítógép családokra

Ebben a részben egy rövid bevezetést adunk 3 kompjúterhez, amiket legtöbbször példának hozunk fel a könyv nagy részében: a Pentium II, az UltraSPARC II és a picoJAVA II[sic].

1.4.1. A Pentium II bevezetés

1968-ban Pobet Noyce, a szilícium integrált áramkör feltalálója, Gordon Moore, aki Moore-törvényéről hírneves, és Arthur Rock, egy San Fransiscoi kapitalista vállalatot alapítottak az Itel Corporation-t, hogy memória csipeket gyártsanak. Működése első évében az Intel csak 3000 dollár értékű csipet adott el, de az üzlet azóta fellendült.

A 60-as évek második felében a számológápek nagy elektromechanikus eszközök voltak, amik mérete megegyezett egy modern lézernyomtatóval és 20 kg-ot nyomtak. 1969 Szeptemberében egy japán vállalat, a Busicom megbízta az Intelt, hogy készítsen 12 csipet egy tervezett elektromos számítógéphez. Az Intel mérnökei nekiláttak a projectnek. Ted Hoff megnézte a terveket és észrevette, hogy bele tud helyezni egy 4 bites általános célú CPU-t egy önálló csipbe, ami ugyanazt a funkciót látja el, de ugyanakkor egyszerűbb és alcsóbb is. Így 1970-ben az első ilyan CPU, ami egyetlen csipben van, a 2300-tranzisztor 4004 meglátta a napvilágot. (Faggin et al., 1996).

Ez semmit sem ért mivel sem az Intel, sem a Busicom nem tudta, hogy épp mit fedeztek fel. Amikor az Intel eldöntötte, hogy megérné esetleg a 4004-est kipróbálni a többi projectben, felajánlotta, hogy visszavásárolja az új csip összes jogát a Busicomtól, visszatérítése 60 000 dollárt, amit a Busicom fizetett neki kifejlesztésért. Az Intel ajánlatát gyorsan elfogadták, aki ezután hozzálátott a csip 8 bit-es változatának kidolgozásához, amit 1972-ben mutattak be.

Az Intel nem várt nagy keresletett a 8008, így egy alacson volumenű gyártásra állt be. De legtöbbek megrökönyödésére akkora volt az érdeklődés, hogy az Intel hozzáfogott egy új CPU csip tervezéséhez, aminek a memóriája a 8008-a 16K-s kapacitásához hasonló (amit a csipen levő láb száma ír elő). Ez a kivitelezés a 8080-t eredményezte, egy kicsi általános célú CPU-t, aminek bemutatására 1974-ben került sor. Sokkal inkább, mint a PDP-8-as, ez a termék egy csapásra meghódította a szakmát, és azonnal tömegcikk lett. Az ezres eladás helyett (amit a DEC tett) az Intel milliókat adott el.

1978-ban jött a 8086, igazi 16-bites CPU egy önálló csipen. Az 8086-ost úgy tervezték, hogy kicsit hasonlítson a 8080-ashoz, de ne legyen vele kimpatibilis. A 8086-ost a 8088-as követte, aminek ugyanaz az architekturája, mint a 8086-nak és ugyanazokat a programokat futtatta, de volt egy 8-bites buszja a 16 bites helyett, ami lassúbbá viszont olcsóbbá tette a 8086-osnál. Amikor az IBM kiválasztotta a 8088-ast, mint az eredeti IBM PC CPU-ját, ez a csip nagyon gyorsan a személyi számítógépipar standardjévé vált.

Sem a 8088-as sem a 8086-os nem tudott 1 Mbyte memóriánál többet küldeni. Az 1980-as évek első felére ez egyre inkább égető problámává vált, tehát az intel megtervezte a 8286-ot, ami a 8086 fölfelé kompatibilis verziója.

***[30-33]

Fordítás: 30-33.oldal
Készítette: Gyarmati Andrea (h938709)

Az alap utasítás készlet lényegében azonos volt a 8086 és a 8088-ban. De a memória kezelés teljesen különbözött, és inkább nehézkes volt. A feltétel a korábbi chippekkel való kompatibilitás volt. A 80286-ost IBM PC/AT gépeken használták és a középkategóriás PS/2-ben. Csak úgy mint a nagysikerű 8088-ast, lényegében azért, mert ezt egy gyorsabb 8088- asnak tekintették. .

A következő lépés az igazi 32 bites processzor volt, a 80386, amit 1985-ben hoztak ki. Mint a 80286-os ez is többé-kevésbé kompatibilis volt visszamenőleg egészen a 8080 -ig . A visszafelé való kompatibilitás elôny volt azok számára, akiknek a régi programok futtatása fontos volt, de ellenszenvett ébresztettek azokban , akik jobb szerették az egyszerű, tiszta, modern a múlt hibáitól mentes felépítést.

Négy évvel később kijött a 80486-os. Ez lényegében egy gyorsított válltozata volt a 80386-nak, aminek szintén volt egy lebegő-pontos-egysége, valamint 8K Cache memóriát is tartalmazott a chipbe építve. A cache memóriát arra használják, hogy tárolja a legtöbbet használt memória részt, a processzorban vagy annak a közelében, a lassú központi memória hozzáférés elkerülése végett. A 80386-os támogatja a többprocesszoros környezetet, ami megengedi a gyártóknak, hogy olyan rendszereket építsenek , amiben több
processzor van.

Ezen a ponton az Intel rájött, hogy a nehéz (mivel elvesztette a szabadalmi pert) szabadalom alá vonni számokat (mint a 80486), így a következő generáció nevet kapott: Pentium (a görög szóbol penta (öt)). A 80486 -ban egy beépített pipeline van , ellenben a Pentiumban már kettő, amelyik lehetővé teszi kétszer gyorsabban való végrehajtást. (A pipeline -ról már volt szó a 2 fejezetben).

Amikor a következő generáció megjelent, elkeseredtek azok, akik abban reménykedtek, hogy az új család a Sexium (latin: sex = 6) nevet kapja. A Pentium név már jól csengett a köztudatban, igy az értékesítéssel foglalkozó emberek meg akarták tartani és az új chip neve a Pentium Pro lett. A kis névváltozás ellenére ez a processor nagy áttörést mutatott. A helyett, hogy kettő vagy több Pipeline lenne benne, a Pentium Pro telljesen más belső szervezése révén 5 utasítást tudott egyszerre végrehajtani.

A másik újjítás , hogy a Pentium Pro két szintes Cache memóriát tartalmazott. Magában a chipben van 8K a gyakran használt utasítások tárolására , és van még 8K a gyakran használt adatok számára. Valamint ugyanabban a tokban (a Pentium Pro dobozában) , de nem magába a Chipbe építve van a második-szintű cache, ami 256 KB.

A következő Intel processzor a Pentium II , lényegében egy Pentium Pro egy különleges multimedia utasítás kiegészítéssel (MultiMedia Extension). Ezek az utasítások a számitás igényes audio és video lejátszások sebességét hivatottak növelni, fölöslegessé téve egy multimédiás társprocesszor hozzáadását. Ezek az utasítások a későbbi Pentiumokban is hozzáférhetőek, de a Pentium Pro-ban nem. Egyszóval a Pentium II kombinálta a Pentium Pro erősségeit egy multimédia támogatással.

1998 elején az Intel bemutatta új termékét a Celeront, ami alapjába véve egy alacsony árú, gyengébb teljesítményű verziója a Pentium II-nek. A Celeront alacsony

minőségű PC-be szánták (low-end). Mivel a Celeron ugyanolyan felépítésű, mint a Pentium II , nem fogjuk megvitatni a könyvben. 1998 júniusában a Pentium II különleges verzióját mutatta be az Intel, amit a piac magasabb rétegeibe szánt. Ez a processzor a Xeon nevet kapta, nagyobb cache, gyorsabb bus, valamint jobb többprocesszoros támogatás jellemzi, de másrészről egy normál Pentium II, így ezt sem beszéljük meg részletesen a későbbiek folyamán.

Chip	**Dátum**	**MHz**	**Tranzisztor**	**Memória**	**Megjegyzés**
4004	4/1971	0.108	2,300	640	Elsőmikroprocessor chipen
8008	4/1972	0.108	3500	16 KB	8 bites mikroprocesszor
8080	4/1974	2	6000	64 KB	Első minden célnak megfelelő
8086	6/1978	5-10	29000	1 MB	Első 16 bites CPU
8088	6/1979	5-8	29000	1 MB	Ezt használták az IBM PC-ben
80286	2/1982	8-12	134000	16 MB	Már van memória védelem
80386	10/1985	16-33	275000	4 GB	Az első 32 bites CPU
80486	4/1989	25-100	1.2 M	4 GB	8K cache beépítve
Pentium	3/1993	60-233	3.1 M	4 GB	2 Pipeline, később MMX
Pentium Pro	3/1995	150-200	5.5 M	4 GB	2 szintű cache beépítve
Pentium II	5/1997	233-400	7.5 M	4 GB	Pentium Pro + MMX

{ Az ábrán az Intel család látható. Az órasebesség MHZ-ban van megadva, ami 1 millió fordulat/másodperc. }

Az összes Intel chip kompatibilis volt az elődeivel vissza egészen a 8086. Egyszóval a Pentium II képes 8086 programokat futtatni módosítás nélkül. Ez a kompatibilitás mindig is tervezési minimum volt, hogy megengedjék a felhasználóknak, hogy megtarthassák a már meglévő szoftver beruházásukat. Természetesen a Pentium II 250 - szer bonyolultabb, mint a 8086, így a Pentium II meg tud csinálni olyan dolgokat, amit a 8086- os nem tudott. Ezek a részekre szedtett darabok azt erdeményezik, hogy a felépítése nem annyira elegáns, mintha odadnánk valakinek a Pentium II -t felépítő 7.5 millió tranzisztort és utasításait, hogy kezdje el elölről fölépíteni.

Érdekes megjegyezni, hogy a Moore törvénye -sokáig a memória bitjeinek számához volt társítva- ugyanolyan jól alkalmazható a processzorokra is. Ábrázolva a trancisztorok számát a megjelenés függvényében, észrevesszük, hogy minden chip egy félegyenesen helyezkedik el; itt is azt látjuk, hogy Moore törvénye igaz.

Az UltraSPARC II bemutatása

Az 1970- es években a UNIX volt népszerű az egyetemeken, de nem a

személyi számítógépeken futó UNIX, tehát a UNIX-kedvelők időosztásos (gyakran túlterhelt) miniszámítógépeket használtak, mint például a PDP-11 és a VAX.

1981-ben egy German Stanford-on végzett hallgató, Andy Bechtolsheim, aki megunta, hogy a számítógépközpontokban UNIX-ot használtak, elhatározta, hogy épít magának egy személyi UNIX munkaállomást használaton kívüli részekből. SUN-1-nek (Stanford University Network) hívta.

Bechtolstheim nemsokára felhívta Vinod Khosla, egy 27 éves indiai figyelmét, aki égett a vágytól, hogy 30 éves korára milliomos legyen. Khosla meggyőzte Bechtolsheim-et, hogy alakítson egy céget, ami SUN munkaállomásokat épít és ad el. Khosla szerződtette Scott McNealy-t, egy másik Stanford-on végzett diákot a gyártás vezetőjének. A szoftver megírásához szerződtették Bill Joy-t, a Berkeley UNIX legfontosabb tervezőjét. Ők négyen alapították a SUN Microsystem-et 1982-ben.

A SUN első terméke Sun-1, amely MOTOROLA 68020 CPU-val működött, amely azonnali siker volt, mint az ezt követő SUN-2 és SUN-3 gépek is, melyek szintén MOTOROLA CPU-t használtak. Nem lehet más személyi száémítógépekkel egy napon említeni. Ezek a gépek messze erősebbek voltak (innen az elnevezés “munkaállomás”), és azt tervezték, hogy a kezdetektől hálózaton fog futni. Mindegyik SUN munkaállomás ETHERNET kapcsolattal és IPC/IP szoftverrel az ARPANET-hez, az Internet előfutárához való kapcsolódáshoz.

1987-re a SUN eladott fél billió dollár értékű rendszereket, elhatározta, hogy megtervezi saját CPU-ját. Ezt egy új forradalmi tervre alapozta a California Egyetemről Berkeley-ből (RISK II). Ez a CPU, amit SPARC-nak (Scalable Processzor Architecture) hívtak, a SUN-4 munkaállomás alapja. Rövid időn belül minden SUN terméket használt a SPARC CPU.

Eltérően a többi számítógépes cégtől, a SUN elhatározta, hogy nem maga fogja gyártani a SPARC CPU chipeket. Ehelyett engedélyezte gyártásukat néhány különböző kisebb vállalatnak remélve, hogy a versengés köztük növelni fogja teljesítményüket és csökkenti az árakat Ezek a vállalatok számos különböző chippeket gyártottak, különböző technológián alapulva, különböző órasebességgel futva és különböző áron. Ezek a chipek magukba foglalták a MicroSPARC-ot, a HyperSPARC-ot, a SuperSPARC-ot és a TurboSPARC-ot. Bár ezek a CPU-k kisebb dolgokban különböztek, de mind kompatibilis volt és futott rajtuk ugyanaz a felhasználói program módosítások nélkül.

A SUN mindig azt akarta, hogy a SPARC nyitott felépítésű legyen sok ellátóegységgel és rendszerrel, azzal a céllal,hogy építsen egy vállalatot, amely versenybe tud szállni személyi számítógépek terén a világon már uralkodóvá vált Intel alapú CPU-kal. Elnyerték a cég bizalmát, amely érdekelve volt a SPARC-ban, de nem akartak beruházni a gyártás ellenőrzésébe, mint vetélytárs. A SUN megalakított egy ipari konzorciumot SPARC International néven, a SPARC felépítésének jövőbeli változatának kifejlesztésére. Így fontos külünbséget tenni a SPARC felépítés, amely egy pontos felvilágosítást ad és más, programban látható jellemvonást tartalmaz, és ezek egy sajátos eszköze között.Ebben a könyvben meg fogjuk tanulni mindegyiket a generikus SPARC felépítéséről, és azután tárgyaljuk a CPU chipeket Chaps-ban. A 3 és a 4 egy speciális SPARC chipet használ a SUN munkaállomásokban.

A kezdeti SPARC 32-bites gép volt, 36 Mhz-n futott. A CPU-t IU-nak (Integer Unit) hívták, egyszerű és átlagos volt a 3 legfontosabb utasítás formátummal és összesen 55 utasítással. Ráadásul a lebegőpontos egység hozzáadott másik 14 utasítást. Ez ellentétben állt az Intel vonalával, amely 8- és 16-bites chppekkel (8086, 8088, 80286) kezdte és végül a 80386-tal 32 bites lett.

A SPARC első megtörése 1995-ben következett be a SPARC felépítésének 9.verziójának megjelenésével, egy 64-bites felépítés, 64-bites címzéssel és 64-bites regeszterekkel. Az első SUN munkaállomás a V9 (Version 9) felépítés volt az UltraSPARC I, amit 1995-ben vezettek be. (Trembley és O`Connor, 1996).
A 64-bites gépek ellenére, ez is teljesen binárisan kompatibilis volt a már létező 32-bites SPARC-okkal.

Az UltraSPARC meg akarta törni a gyanut. Ezzel szemben az előző gépekben meg volt tervezve, hogy alfanumerikus adatot kezel és a futó programok, mint word processzorok, vírusirtás, az UltraSPARC-ban általában meg volt tervezve a kezdetektől a képek, audio, video és multimédia kezelése. Más újítások között a 64-bites felépítés mellett volt 23 új utasítás,magába foglalt néhány tömörített és nem tömörített képelemet 64-bites szavaktól, képek kicsinyítését és nagyítását, blokkok mozgatását, video ki-és betömörítését. Ezeknek az utasításoknak, röviden VIS (Visual Instruction Set) az a céljuk, hogy általános multimédia hozzáférhetőséget biztosítsanak az Intel MMX utasításokhoz hasonlóan.

Az UltraSPARC olyan magasszintű alkalmazásokra van tervezve, mint a nagy multiprocesszoros Web szerverek tucatnyi CPU-val és 2TB (terabyte)-ig terjedőfizikai memóriával. Mindazonáltal kisebb verziók notebook számítógépeken is használhatók.

Az UltraSPARC I örököse az UltraSPARC II és az UltraSPARC III volt. Ezek főleg az órasebességben különböznek, de néhány új jellemzőt is hozzáadtak minden új változathoz.

***[34-37]

Hozzá voltak adva az ismétlések is. Ebben a könyvben ,amikor megvitatjuk a SPARC sturkturát,
Használni fogunk 64-bites V9 ULTRASPARC II –t úgy mint példát.

1.43 Beveteés a picoJava II –be

A C programnyelvet Denis Ritchie a Bell labóratórium dolgozója találta fel a Unix operációs rendszer számára . A Unix gazdaságos tervénak és népszerűségének köszönhetően ,a C hamarosan domináns programnyelv lett. Néhány év múlva a Bell labóratóriumos Bjarne Stroustrup hozáadott néhány ötletet
Az objectum orientált programozás világából a C-hez, hogy létrehozza a C++-t ,amely szintén nagyon népszerű lett.

A 90-es évek közepén a Sun microsystem kutatói módokat kerestek arra, a felhasználók bináris programokat szedjenek le az Interet-ről, és World Wide Web (www) oldalként futassák.
Szerették a C++ -t ,de nem volt elég biztos. Más szóval egy új C++ bináris program könnyen
″ belemászhatott ″ a gépekbe. A megoldás egy új nyelv feltalálása volt a Java-é, amit a C++ ihletett,
de biztonsági problémák nélkül. A Java gépelés biztos nyelv melyet egyre többen használtak. Mivel népszerű és elegáns nyelv, használni fogjuk a könyvünkben programpéldákra.

Mivel a Java csak programozási nyelv, lehetséges szerkesztőket írni a számára Pentium-ra,
Sparc-ra vagy ás architecturákra. Ilyen szerkesztők számára.
A Sun fő célja a Java-val az volt, hogy lehetővé tegye a futtatható programok seréjét az Internetes
És hogy a felhasználó módosítás nélkül futtathasson. Ha egy SPARC-on szerkeztett Java programot
Az interneten átküldenék egy Pentiumnak, nem futna, meghamisítva a célt, hogy képes bináris programot küldeni bárhová, és ott futtatni.

Hogy a bináris programotkat hordozhatóvá tegyük különböző gépeken, a Sun meghatározott egy virtuális gépszerkezetet JVM (Java virtual machine) néven.
Ennek a gépnek 32 bites szavakból és 226 parancsból álló memóriája van. ezek többnyire egyszerűek,
De néhány komplexebb, többszörös memória aktust igényel.

Hogy a Java-t hordozhatóvá tegye, a Sun írt egy szerkesztőt, ami összeköti a Java-t a JVM-mel
Írt egy JVM fordítóprogramot is a Java bináris programok futtatására. Ezt C nyelven írta , ezért C szerkesztős gépen futtatható , azaz majdnem minden létező gépen . Tehát ha Java bináris programokat akarunk futtatni , csak egy futtatható bináris programot kell szereznünk a JVM fordítónak amelynek felülete (Pentium II és Windows 98 ,Sparc ,UNIX stb) lehet bizonyos támogatott programokkal és könyvtárakkal. A legtöbb Internetböngésző kalauzhoz van JVM fordító, hogy megkönnyítse az **applet**

Futtatását , amik kicsi Java bináris programok , Társítva a WORD WIDE WEB oldalakkal. Sok közöttük amely animációt és hangot is produkál.

A JVM programok (vagy más programok, azért a dolgokért) fordítása lassú. Egy alternatíva a az **applet** futtatására : először kézzel megszerkeszteni a JVM programot a gépen és aztán futtatni.
Ez a stratégia megkívánja a JVM gépnyelv szerkesztő meglétét a böngészőben és azonnali futtathatóságát.Az ilyen fordítókat JIT –nek (Just it time) hívják. De nagy helyet foglalnak és amig lefordítja a programot a JVM gép nyelvére sok idő telik el.

A JVM szoftverek mellett (JVM fordító és JIT szerkesztő), a Sun és más cégek is terveztek hardwert. JVM chipekkel. Ezek a CPU–k amelyek közvetlenül futtatják a JVM bináris programokat.
A kezdeti architecturák, az eredeti picoJava I (O'Connor és Tremblay 1997) és picoJava II (McGhan és O'Connor 1998), a beágyazott eszközök piacát célozzák meg. Ez a piac erős, flexibilis és olcsó chipeket
Igényel (50 $ vagy alatta) amik kártyákba vagy Tv –kbe telefonokba és más eszközökbe vannak ágyazva, főleg kommunikációra használatos dolgokba. A Sun licenszei saját chipeket tudnak gyártani a
PicoJava design-t használva a cahce méretét változtatva, hozzátéve vagy elvéve a Floating –point egységeket.

A Java chip értéke a piacon az, hogy meg tudják változtatni a funkcióját.Pl : vegyünk
Egy üzletembert aki eddig nem olvasott faxot a telefon kis képernyőjén, de egyszer szüksége lesz rá.
Az ellátot híva letöltheti a Faxnézőt a telefonon. A memória hiánya a JIT szerkesztését lehetetlenné teszi. Ebben a helyzetben JVM chip hasznos.
Bár a picoJava II nem egy konkrét chip (nem megvásárolható), ez számos más chip alapja .
Pl : Sun microJava 701 processzoré (Cpu) és a különböző chipektől a Sun licensig. Mi a picoJava II fogjuk használni mert nagyon különbözik a Pentium II-től és az Ultrasparc processzoroktól (CPU) és
Más területen használjuk. Ez a processzor érdekes a mi céljainkra nézve , mert a IV. fejezetben

Pl : JVM microprogram-ot jól fogjuk használni. Mi viszont tudni fogjuk hogy a mictoprogramunk design-ja ellentetje a igazi hardwere design-jának

A picoJava II két része van a Cache és a flooating –point egység, amelyet mindenki jól eltávolíthat. Az egyszerűség kedvéért a picoJava II –re chip -ként és nem chip design -ként fogunk utalni. Néha utalni fogunk a Sun microJava 701–es chipre, ami végrehajtja a PiciJava II designt.

A Pentium II –t, ULRASPARC –t és a picoJava II használva 3 féle processzort (CPU) tanulmányozunk. Ezek hagyományos CISC felépítésűek modern SUPERSCALAR technológiával felszerelve, a Risc felépítés azt jelenti, hogy superscalar technológia és a java chip be van, ágyazva a rendszerbe. Ez a három processzor nagyon különbözik egymástól, alkalmat adnak nekünk megvizsgálni

A jobb űrtechnológiát és láthatjuk amit a sokféle optimalizálássokkal lehet készíteni processzorokat .

1.5 Miről szól a könyv

Ez a könyv a multilevel computerekről szól.
4 szintet (level -t) fogun vizsgalni.(Digitális logika, operaciós rendszer, microarchitectura , Isa).
Fő témáink A memória
A parancsok
A design.
Ezeknek a tanulmányozását a computer organization -nek nevezik.
Elsősorban fogalmakkal fogunk foglalkozni, ezért a példák le lesznek egyszerűsítve, hogy a lényegre koncentráljunk.
Hogy betekintést nyerjünk abba, hogy az elméletet hogyan használjuk a gyakorlatban.
Használni fogjuk a Pentium II-t ,ULRASPARC-ot és a picoJava II-t.
Több okból választottuk ezeket.

I. Széles körben használják őket az olvasó biztos, hogy fog találkozni valamelyikkel.
II. Mindkettőnek megvan az egyéni felépítése, amely alapot nyújt az összehasónlításra.

Az olvasót próbáljuk ösztönözni a számítógép kritikus (alpos) vizsgálatára.
Szeretnénk leszögezni, hogy ez a könyv nem a Pentium II, ULTRASPARC II, picoJava II programozásáról szól.
A II. fejezet bevezetés a computer alapvető részeibe. Szeretnénk áttekintést nyújtani a könyv felépítésében és
Bevezetőt adni a további fejezetekbe.

III, IV, V, VI Fejezetben : A LEVEL-ek (szintekről) szól.
A k level design-ja a k-1 level által meghatározva, tehát csak akkor érthető, ha tisztában vagyunk az azt motiváló level-kel.

III. **F**ejezet: A digital logic level (digitális logikai szint)
Mi a gate (kapu).
Boolean algebra.
PCI Bus
IV. **F**ejezet: A microarchitecture level –ről szól.
V. **F**ejezet: ISA level –ről szól.
VI. **F**ejezet: Parncsok
Memória
Példák: A Unix és Windows NT-ben, ULTRASPARC-N.
VII. **F**ejezet: Nyelvi level.
VIII. **F**ejezet: Párhuzamosan használt komputerek.
IX. **F**ejezet: Ajánlott olvasmány.

PROBLÉMÁK

Magyarázd meg saját szavaiddal a következőfogalmakat:

a. Translator (Transzformátor).
b. Interpeter (fordító)

c. Virtual machine (virtuális gép)

***[38-41]

2. .Mi a különbség az értelmezés és fordítás között?
3. Elképzelhető-e egy fordító részéről, hogy a mikro architektura szintjén generáljon outputot az ISA szint helyett? Vitassák meg a feltevés mellett és ellen szóló érveket!
4. El tud képzelni egy többszintű számítógépet, amelyben az eszköz szint és a digitális logika szint nem a legalacsonyabb szintek? Magyarázza meg!
5. Vegyünk egy számítógépet, melynek az értelmezője megegyezik az 1., 2. és 3. szinten. Egy értelmezőnek n utasításra van szüksége ahhoz, hogy egy utasítást elszállítson, megvizsgáljon és végrehajtson. Az első szinten egy utasítás k nanosecundum alatt hajtódik végre. Ez mennyi időbe telik a 2., 3., 4. szinteken?
6. Vegyünk egy többszintű számítógépet, ahol mindegyik szint különbözik. Mindegyik szintnek vannak olyan utasításai, melyek m-szer erősebbek., mint az alatta lévő szinten levők, ami azt jelenti, hogy egy r szintű utasítás r-1 db m szintű utasítás feladatát látja el. Ha egy első szintű programnak k másodperc szükséges a futáshoz, mennyi időbe kerülne egy ugyanolyan programnak a 2., 3., 4. szinten, feltételezve, hogy n szintű r utasítás szükséges ahhoz, hogy értelmezzen egyetlen r+1 utasítást?
8. Milyen értelemben ekvivalens a software és hardver? Milyenben nem?
9. Neumann azon ötletének, hogy a programokat a memóriában tároljuk, egyik következménye, hogy a programok az adatokhoz hasonlóan megváltoztathatóak. Tudna egy olyan példát említeni, melyikben ez az adottság hasznos lehet. (tipp: Gondoljon az aritmetika használatára a tömbökön!)
10. A 360 modell 75 teljesítményarány 50- szerese a 360 modell 30, de a ciklusidő csak ötször olyan gyors. Hogyan indokolja ezt az ellentmondást?
11. Két alap rendszerterv látható az 1-5 és 1-6 ábrán. Írja le, hogyan történhet az input/output ezen rendszerekben. Melyiknek van lehetősége a jobb rendszerteljesítményre?
12. Egy bizonyos időpontban egy tranzisztor, egy mikroprocesszor 1 micron volt diaméternben. Moore törvénye szerint a következő évi modellben milyen nagy lenne egy tranzisztor?

2
Számítógéprendszerek felépítése

Egy digitális számítógép processzorok, memóriák, ki- és bemeneti egységek (I/O) kölcsönösen összekapcsolt rendszeréből áll. Ez a fejezet bevezetés ehhez a három összetevőhöz, valamint háttér a következő öt fejezet specifikus szintjeinek részletes megvizsgálásához. A processzorok, memóriák, a ki- és bemeneti egységek kulcsfogalmak, és minden szinten jelen lesznek, így azzal kezdjük számítógép architektúra tanulmányainkat, hogy megnézzük ezt a három egységet.

2.1 Processzorok

A 2-1. ábra mutatja egy egyszerű busz orientált számítógépes felépítését. A CPU (központi vezérlő egység) a számítógép "agya". Funkciója az, hogy a main memóriában tárolt programokat végrehajtsa; elérve azok utasításait, megvizsgálja, majd egymás után végrehajtja őket. Az összetevőket egy busz kapcsolja össze, ami cím, adat és ellenőrzőszignálok átvitelére való párhuzamos kábelek gyűjteménye. A buszok lehetnek CPU-n kívüliek, a memóriához és az I/O eszközökhöz kapcsolva, de

lehetnek CPU-n belüliek is, mint ahogyan ezt hamarosan látni is fogjuk.
A CPU számos elkülönülő részből áll össze. A vezérlőegység felelős az utasítások elszállításáért a main memóriából, valamint a típusaik meghatározásáért. Az aritmetikai logikai egység (ALU) olyan műveleteket végez, mint az összeadás, vagy a Boolean típusú AND művelet, amelyek szükségesek az utasítások végrehajtásához.
A CPU ezen kívül még tartalmaz egy nagy sebességű memóriát, amelyet ideiglenes eredmények és bizonyos vezérlőinformációk tárolására használ. Ez a memória számos regiszterből áll, amelyeknek meghatározott méretük és funkciójuk van. Általában az összes regiszter ugyanolyan méretű. Mindegyik regiszter egy számot tud tárolni, maximálisan a regiszter mérete alapján meghatározottat. A regisztereket gyorsan lehet olvasni és írni, mivel ezek a CPU belsejében találhatóak.
A legfontosabb regiszter a programszámláló (PC), amelyik a következő végrehajtandó elszállítandó utasításra mutat. A "programszámláló" név kissé félrevezető, mert a számoláshoz semmi köze, mégis ezt a kifejezést használják rá általánosan. Ugyancsak fontos az utasításregiszter, amely az éppen végrehajtás alatt lévő utasítást tartalmazza.
A legtöbb számítógépnek számos más regisztere is van. Néhány ezek közül általános, néhány specifikus célokat szolgál.
Ábra 2-1.: Egyszerű számítógép felépítése egy CPU- val és két be/kimeneteli egységgel.

2.1.1 A CPU felépítése

Egy tipikus Neumann-féle CPU belső szerkezetének részeit részletesebben a 2-2. ábra mutatja. Ezt a részt data pathnak hívják, és a regiszterekből (általában 1-től 32-ig),az ALU-ból és néhány buszból áll, amik összekapcsolják a darabokat. Ezek a regiszterek táplálják a két ALU input regisztert, amit az ábra A-val és B-vel jelöl. Ezek a regiszterek tárolják az ALU inputját, amíg az ALU számol. A data path minden gépben nagyon fontos, így a könyvben részletesen fogunk majd foglalkozni vele.
Az ALU saját maga végzi az összeadást, a kivonást és más egyszerű műveleteket az inputjain, így adva át az eredményt az output regiszternek. Ezt az output regisztert egy regiszterbe tudjuk besorolni. Szükség esetén ezt a regisztert később a memóriába át lehet írni. Nem minden rendszer rendelkezik az A, B és output regiszterekkel. A példában az összeadást illusztráljuk.
A legtöbb utasítást a következő két kategória egyikébe tudjuk besorolni: regiszter - memória, regiszter - regiszter utasítás. A regiszter - memória utasítás megengedi a memória szavainak, hogy a regiszterekbe szállítódjanak, ahol pl. az ALU inputjaiként lehet őket használni későbbi utasításokban. (a "szavak" a memória és a regiszter között mozgó adategységek. A szó jelölhet egy integer típusú számot is. A memória felépítését később tárgyaljuk meg ebben a fejezetben.) Más regiszter - memória utasítások megengedik a regisztereknek, hogy tárolódjanak a memóriában.
Az utasítás másik fajtája a regiszter - regiszter típusú. Egy tipikus regiszter - regiszter utasítás két operandust szállít a regiszterekből, elviszi őket az ALU input regisztereihez, végrehajt rajtuk pár műveletet, pl. az összeadást vagy a Boolean And műveletet, és az eredményt a regiszterek egyikében tárolja. A két operandus ALU-n keresztüli futtatásának és az eredmény tárolásának folyamatát "data path cycle"- nek hívjuk. Ez a legtöbb CPU lelke. Ez nagyban meghatározza, hogy mit tud egy gép csinálni. Minél gyorsabb a data path cycle, annál gyorsabb a gép.

***[42-45]
Nem készült el! Hodur Albert: h938358

***[46-49]

Bartalos Jenő kpmI. (h734573)

2.1.3. RISC kontra CISC

A hetvenes évek vége felé rengeteget kísérleteztek nagyon összetett utasításokkal, amit az interpreter tett lehetővé. A tervezők próbálták csökkenteni a szakadékot a gépek tudása és a magas szintű programnyelvek által megkívánt fejlettségi szint között.Alig volt valaki, aki egyszerűbb gépek tervezéséről gondolkodott, épp úgy, mint ma, nem sokan terveznek egyszerűbb operációs rendszereket, hálózatokat, szövegszerkesztőket, stb... (talán nem szerencsésen).

John Cocke, az IBM munkatársa vezette azt a csoportot, amely szakított az addigi irányzattal, és megpróbált megvalósítani néhányat Seymour Cray ötleteiből egy nagy teljesítményű mini computerben. Ez a munka egy kísérleti mini-számítógéphez vezetett, amelyet 801-esnek neveztek. Habár az IBM sohasem dobta piacra a gépet, és az eredményeket sem közölték évekig (Radin, 1982), a hír mégis kiszivárgott, és mások is elkezdtek foglalkozni hasonló architektúrák vizsgálatával.

1980-ban egy, a Berkley egyetemen David Patterson és Carlo Séquin által vezetett csoport olyan VLSI CPU chipeket kezdett tervezni, amelyek nem használtak interpretereket. (Patterson, 1985; Patterson és Séquin, 1982). Ők alkották meg erre a koncepcióra építve a RISC elvet. CPU chipjüket RISC I - nek nevezték, amelyet hamarosan a RISC II. követett. Nem sokkal később, 1981-ben, a San Francisco-i öblön túl, Stanfordban, John Hennessy egy kicsit másmilyen gépet tervezett és készített el, amit MIPS- nek nevezett.(Hennessy, 1984). Ezek a chipek fontos kereskedelmi termékekké váltak, a SPARC és a MIPS különösen.

Az az új processzorok jelentősen kőlönböztek az akkoriban kereskedelemben lévőktől. Mivel nem voltak velük visszafelé kompatibilisek, így a tervezők szabadon alkalmazhattak új utasítás-csomagokat, amelyekkel ki lehetett használni a rendszerek teljes kapacitását. Míg kezdetben a hangsúly a gyorsan végrehajtható, egyszerű parancsokon volt, hamar felismerték, hogy az utasítások fejlesztését gyorsan kell végezni, hisz ez a kulcs a sikerhez. Kevésbé számított, hogy milyen hosszú egy utasítás, mint hogy másodpercenként hányat lehet végrehajtatni belőle.

Ezen egyszerű processzorok tervezése idején a figyelem a viszonylag kis számú (kb.50) végrehajtható utasítások felé fordult. Ez a szám sokkal kisebb volt, mint a bevált gépeken kiadható 200-300 parancs.(Ilyen volt pl.: a DEC VAX és az IBM nagy gépei). Igaz, a RISC betűszó Redukált Utasításkészletű Számítógépet jelent (Reduced Instruction Set Computer), ami a CISC ellentéte, mivel ez Összetett Utasításkészletű Számítógépet jelent. (Complex Instruction Set Computer) /ez egy alig leplezett utalás a VAX-nak, amely ez idő tájt uralkodó a Számítástechnikai

Tudományegyetemek között./ Manapság már csak néhány ember gondolja azt, hogy az utasításkészlet mérete egy hatalmas tömb, de a név mégis megmaradt.

Hogy ne nyújtsuk hosszúra a történetet, végül is ádáz háború robbant ki : a RISC támogatói támadást indítottak a biztos alapokon nyugvó többiek (VAX, Intel, IBM) gépei ellen. Azt akarták, hogy a számítógépek tervezése a legjobb úton haladhasson, ez pedig a kis számú, egyszerű utasítások egy ciklusban, a megadott sorrendben történő végrehajtását jelenti (2-2 ábra).

47.

Nevezetesen, vegyünk két regisztert, egyesítsük őket valahogy (pl. adjuk vagy kapcsoljuk össze őket), és tároljuk a kapott eredményt egy regiszterben. Az eredmény a következő: Egy RISC elven működő gép 4-5 utasítással hajtja végre azt, amit egy CISC- gép egyetlen paranccsal elintéz, azonban tizszer gyorsabban dolgozik.(hiszen nincs interpretálva) Így a RISC győz. Arra is érdemes rámutatni, hogy ez idő alatt a fő memória sebessége utoléri a csak olvasható, irányító tár sebességét. Az interpetáció hátránya így még inkább nő, ami igen kedvező a RISC gépek számára.

Logikus, hogy az ilyen előnyöket élvező RISC technológia gépei (pl. DEC Alpha) ki kellene, hogy szorítsa a piacról a CISC gépeit (pl. Intel Pentium). Semmi ilyesmi nem történik. De miért?

Először is itt van a visszafelé kompatibilitás ténye. A vállalatok dollármilliókat invesztáltak az Intel-irányvonal szoftvereibe. Másodszor pedig, meglepő, de az Intel képes volt alkalmazni ugyanazokat az ötleteket CISC környezetben is.A 486-osoktól kezdve az Intel CPU-k tartalmaznak egy ún. RISC-magot, a legegyszerűbb (és általában leggyakoribb) parancsokat megadott sorrendben, egy cikluson belül hajtja végre. Ez alatt pedig a szokásos CISC módszerrel interpretálja az összetettebb utasításokat. Hálózaton használva az eredmény azt mutatja, hogy az egyszerű utasítások gyorsak, a bonyolultabbak lassúak. Ugyan ez a hibrid megoldás nem olyan gyors, mint a teljes RISC konstrukció, de versenyképes, amíg a régi szoftvereket módosítás nélkül engedik futni.

2.1.4. A modern számítógépek konsrukciós elve

Több, mint egy évtized telt el azóta, hogy az első RISC gépek bemutatkoztak.
Egyes konstrukciós elvek elfogadottá váltak, mint a számítógépek fejlesztésének helyes irányvonalai, s elterjedtek a hardver technológiában. Ha nagyobb változás köszöntene be (pl. egy új ipari eljárás következtében a memória elérési ideje hirtelen a CPU elérési idejének tizedére csökkenne), minden felborulna. Így a tervezőknek mindig figyelemmel kell kisérniük a technikai változásokat, melyek befolyásolhatják az összetevők közötti egyensúlyt.

Tehát, a konstrukciós elveknek rendszere van, ezt néha RISC konstrukciós elveknek is hívják, ezek az általános CPU-k azon felépítései, amelyek a tőlük telhető legjobbat adják, hogy versenyben maradhassanak. A külső kényszerek, mint pl. az az elvárás, hogy néhány meglévő architektúrával megmaradhasson a visszafelé kompatibilitás, sokszor, időről időre kompromisszumokhoz vezetnek. Ám ezek az

elvek olyan "vereségek", amelyekkel a legtöbb tervezőnek meg kell küzdenie. Ezen nyomások alatt fogjuk megalkotni a legnagyobb dolgokat.

A hardver közvetlenül hajtja végre az összes utasítást

Minden egyszerű parancsot közvetlenül végrehajt a hardver. Nincsenek mikroutasítások által értelmezve. Az interpretáció szintjén történő kiküszöbölés a legtöbb utasítás számára nagy sebességről gondoskodik.

Bartalos Jenő kpmI. (h734573)

A computerek számára ezt a CISC utasításkészletei biztosítják. A bonyolultabb műveletek különálló részekre bonthatók, amelyek mikroutasítások sorozataként hajtathatók végre. Ez az extra lépés ugyan lelassítja a gépet,de a ritkábban előforduló utasítások esetén elfogadható.

A kiadott parancsok arányának maximalizálása

A modern számítógépek rengeteg trükköt alkalmaznak, hogy maximalizálhassák teljesítményüket. Ezek közül a legfontosabb, hogy a gép megpróbál annyi utasítást kiadni másodpercenként, amennyit csak lehetséges. Tehát, ha ki tudsz adatni másodpercenknt 500 millió utasítást, akkor egy 500 MIPS-es processzort építettél, ahol nem számít hogy meddig tart míg az aktuális utasítás befejeződik. (a MIPS betűszó millió utasítás per másodpercet jelent.) Ez az elv azt a párhuzamosságot sugallja amely nagy szerepet játszhat a teljesítmény fejlesztésében, mivel nagy számú lassú utasítások kiadása rövid idő alatt csak akkor volna lehetséges, ha a sokféle parancsot egyszerre lehetne végrehajtatni.

Habár az utasítások a programrendszerben mindig összefutnak, mégsincsenek mindig kiadva (hiszen néhány szükséges forrás foglalt lehet), és nem szükséges az sem, hogy itt végződjenek. Természetesen, ha parancs1 -et egy regiszterbe helyezzük, és parancs2 is ugyanazt a regisztert használja, akkor nagy munka meggyőződni arról, hogy parancs2 mindaddig nem olvassa el a regisztert, míg az a helyes értéket tartalmazza. Ezek a nagy igények sok-sok könyveléssel járnak, de a lehetőség megvan a teljesítménybeli előny megszerzésére: a sokféle utasítást egyszerre kell végrehajtani.

Egyszerű kell legyen az utasítások dekódolása

Az utasítások végrehajtási arányában van egy kritikus határ. Ez megmutatja, hogy a parancsoknak milyen forrásokra van még szükségük ahhoz, hogy végrehajtathatók legyenek. Bármi, ami ezt az eljárást segíteni tudja hasznos. Ez magában foglalja, hogy szabályos, meghatározott hosszúságú és kis helyet foglaló utasításokat kell készíteni. A legjobb, ha ezek formátumukban teljesen eltérnek egymástól.

A memóriát csak töltéskor és tároláskor kellene igénybe venni

Az egyik legegyszerűbb mód, hogy egy műveletet különálló lépésekre bonthassunk az, ha megköveteljük, hogy a jel a legtöbb utasítás számára regiszterekből jöjjön és oda menjen vissza. Különálló utasításokkal meg lehet valósítani a műveleti jelek mozgatását a memória és a regiszterek között. Míg a belépés a memóriába hosszú időt igényelhet, és a késés kiszámíthatatlan, ezek az utasítások könnyen egymásba csúszhatnak, ekkor pedig semmi más nem történik csak ide-oda szaladgálnak a jelek a memória és a regiszterek között. Az eredmények azt mutatják, hogy csak a TÖLTÉS és TÁROLÁS parancsoknak kellene használni a memóriát.
Bartalos Jenő kpmI. (h734573)

Sok regiszterről gondoskodjunk

Mivel a belépés a memóriába viszonylag lassú, sok regiszterről kell gondoskodnunk
(min 30-ról), így tehát, ha elökerül egy szó, mindaddig tárolhatjuk egy regiszterben, amíg az nagyobb helyet nem igényel. Nem kívánatos dolog kifogyni a regiszterekből, és visszaömleszteni tartalmukat a memóriába, hogy majd később újraindíthassuk őket. Ha csak lehet, kerüljük ezt el! A megvalósítás legjobb módja az, ha rendelkezünk elegendő regiszterrel.

2.1.5. Az utasítás-szintű párhuzamosság

A számítógép fejlesztők mindig arra törekszenek, hogy növeljék az általuk tervezett gépek teljesítményét. Az egyik módszer a chipek gyorsítására, hogy megnövelik a belső óra sebességét. Ám minden új technikának meg van a maga határa, amit a kor pillanatnyi szintje határoz meg. Összefoglalva a legtöbb computer-fejlesztő a párhuzamosság felé törekszik (két vagy több dolog egyidejű elvégzése). Ez az a módszer, amely még inkább növeli a teljesítményt egy adott óra-sebesség mellett.

A párhuzamosságnak két fontos fajtájáról beszélhetünk:az utasítás-szintű, és a processzorok szintjén levő párhuzamosságról. Az előbbiben a gép azt hasznosítja, hogy a párhuzamosság miatt másodpercenként több utasítás adható ki. Az utóbbiban pedig összekapcsolt CPU-k dolgoznak együtt ugyanazon a problémán. Mindkét megközelítésnek meg van a maga érdeme. Ebben a részben az utasítás-szintű paralellizmussal foglalkozunk, a következőben pedig a processzor-szintűt tárgyaljuk.

Csővonal

Évek óta ismert, hogy a memóriából aktuálisan előkerülő utasítások nagy torlódást idéznek elő, s ezzel lassítják a végrehajtási sebességet. Hogy kezelni tudják ezt a problémát, a computereknek vissza kell térnie legalább az IBM-Kibővitésig (1959), amikor még meg volt az a tulajdonság, hogy az utasítások előre fel voltak hozva a memóriából, így kéznél voltak amikor csak szükség volt rájuk. Ezek a parancsok egy regiszterblokkba - az előtárba -voltak behelyezve. Így, amikor szükség volt egy utasításra, rendszerint elő lehetett venni az előtárból - inkább, mint arra várni, hogy feljöjjön a memóriából.

Ennek eredményeképp az utasítások "származása" alapján történő csoportosítás a végrehajtást két részre bontja: előhozási és aktuális végrehajtás. A csővonal koncepciója még tovább viszi ezt a stratégiát. A parancsok végrehajtásának két részre osztása helyett gyakran sok részre osztották, s mindegyiket a hardver meghatározott része irányította. Mindet ami képes párhuzamosan futni.

A 2-4-es ábra az öt egységes csővonalat ábrázolja, amit állványoknak is neveznek. Az állvány1 az utasításokat a memóriából tölti le, és egy tárba helyezi be amíg szükséges. Az állvány2 dekódolja a parancsokat, meghatározza a tipusukat, és hogy milyen jelre van szükségük. Az állvány3 lokalizálja és előhozza a jelet akár a regiszterekből, akár a memóriából.

***[50-53]

Valójában a 4. szakasz hajtja végre az utasításokat, jellegzetesen futtatva az operandusokat az információ útvonalán a 2-2. ábrán. Végül az 5. szakasz visszaírja az eredményt a megfelelő regiszterbe.

utasítást lekérdező egység -> utasítást dekódoló egység -> operandus lekérdező egység -> utasítást végrehajtó egység -> visszaíró egység

2-4. ábra. (a) 5 szakaszos pipeline. (b) Mindegyik szakasz állapota az idő függvényében. 9 ciklust ábrázoltunk.

A 2-4. ábrán láthatjuk, hogy hogyan működik a pipeline az idő függvényében. Az első ciklus alatt S1 dolgozik az 1. utasításon, lekérdezi a memóriából. A 2. ciklus alatt S2 dekódolja az 1. utasítást, mialatt S1 lekérdezi a 2. utasítást. A 3. ciklus alatt S3 lekérdezi az operandusokat az 1. utasításhoz, S2 dekódolja a 2. utasítást és S1 lekérdezi a 3. utasítást. A 4. ciklus alatt S4 végrehajtja az 1. utasítást, S3 lekérdezi az operandusokat a 2. utasításhoz, S2 dekódolja a 3. utasítást és S4 lekérdezi a 4. utasítást. Végül az 5. ciklusban S5 visszaírja az 1. utasítás eredményét, míg a többi szakasz a további utasításokon dolgozik.
Gondoljunk ki egy analógiát, hogy a pipelineok fogalmát tisztábbá tegyük. Képzeljünk el egy tortagyárat, ahol a sütés és a szállítás el van különítve. Tegyük fel, hogy a szállító részlegnek van egy hosszú futószalagja mellette sorban 5 munkással (feldolgozó egységek). Minden 10 másodpercben (a ciklus) az 1. munkás egy üres dobozt rak a szalagra. Ezután a doboz a 2. munkáshoz kerül, aki belerak egy tortát. Egy kicsivel később a doboz megérkezik a 3. munkáshoz, aki lezárja. Utána folytatja útját a 4. munkáshoz, aki rak egy címkét a dobozra. Végül az 5. munkás leveszi a dobozt a szalagról és egy nagyobb tárolóegységbe rakja az áruházba történő későbbi szállításához. Alapvetően így működik a számítógépnél a pipeline: mindegyik utasítás (torta) végig megy néhány feldolgozó lépésen, mielőtt végrehajtva kijut a végén.
Visszatérve a pipelinera a 2-4. ábrához feltételezzük, hogy a ciklusidő 2 nsec volt. Így 10 nsec-be kerül egy utasításnak, hogy végighaladjon az 5 szakaszos pipelineon. Első pillantásra ha egy utasításnak 10 nsec-ig tart, akkor úgy tűnik, hogy a gép 100 MIPS-el fut, de valójában ennél sokkal jobb. Minden ciklusban (2 nsec) egy új utasítást hajt végre, így a feldolgozás mértéke 500 MIPS, nem pedig 100 MIPS.
A pipeline megengedi, hogy a "lappangást" (mennyi ideig tart az utasítást végrehajtani) és a processzor sávszélességét (a CPU MIPS teljesítménye) felváltva használja. Egy T nsec-es ciklusidővel és n szakaszos pipeline-nal a lappangási idő nT nsec és a sávszélesség 1000/T MIPS. (logikus, mivel az időt nsec-ban mérjük, így a CPU sávszélességét BIPS-ben vagy GIPS-ben kellene mérni, de mivel ezt senki sem teszi, így mi sem fogjuk)

Szuperskalár architektúrák

Ha egy pipeline jó, akkor kettő biztos jobb. Egy, a 2-4-es ábra alapján lehetséges

dual- pipeline CPU látható a 2-5. ábrán. Itt egy utasítást lekérdező egység párokban kérdezi le az utasításokat, mindegyiket külön pipelineon indítja el és az ALU-val párhuzamos műveletként hajtja végre. Hogy a párhuzamos futtatás létrejöhessen, a két utasítás nem ütközhet az erőforrások kihasználásában (pl. regiszterek) és egyik sem függhet a másik eredményétől. Egy pipeline-nal a fordítóprogramnak tartania kellett a helyzetét (a hardware nem ellenőrzi és helytelen eredményt ad, ha az utasítás nem kompatibilis) vagy pedig a hibát észreveszi és megoldást talál rá a végrehajtás alatt úgy, hogy extra hardwaret vesz igénybe.

2-5.ábra: dupla 5 szakaszos pipeline egy közös utasítást lekérdező egységgel

Bár az egyszerű vagy dupla pipelineokat legtöbbször RISC-gépeken használják (a 386-osnak és az elődeinek még nem volt egy se), a 486-tól kezdve az Intel elkezdte bevezetni a pipelineokat a CPU-ba. A 486-osnak 1 pipelineja volt és a pentiumnak két 5 szakaszos, durván úgy, mint a 2-5. ábrán, bár a szakaszok beosztása a 2-es és a 3-as között (dekódoló 1-nek és dekódoló 2-nek hívták) nagy mértékben eltért, mint a példánkban. A fő pipeline, amit U- pipeline-nak hívtak végre tudott hajtani egy tetszőleges pentium-utasítást. A második pipeline (V pipeline) csak egy egyszerű, egész számos utasítást tudott végrehajtani (és egy egyszerű lebegőpontos utasítást - FXHC)
A komplex előírások meghatározták, hogy egy utasításpár kompatibilis volt-e, így végre lehet-e hajtani párhuzamosan. Ha az utasítás nem volt elég egyszerű vagy nem volt kompatibilis, a párban csak az első hajtódott végre (az U pipeline-ban). A másodikat fenttartotta és párosította a következő utasítással. Az utasításokat mindig sorrendben hajtotta végre. Ekképpen pentiumra írt fordítóprogramok kompatibilis párok alkotásával gyorsabban futó programokat tudtak produkálni, mint a régebbi fordítóprogramok. A mérések azt mutatták, hogy egy pentiumra optimalizált kód futási sebessége egész számos programok esetében pontosan kétszerese volt, mint egy ugyanolyan órajelű 486-on. (Pountain, 1993). Ez a növekedés teljes egészében a két pipelinenak volt tulajdonítható.
Még a 4 pipeline is elképzelhető, de a hardwaret ugyanígy megduplázza. Helyette más megközelítést alkalmaznak a csúcskategóriás CPUkban. Az alapötlet az, hogy ezek egy pipelinet használnak, de több funkciójú egységeket alkalmaznak, mint a 2-6-os ábrán látszik. Pl. a pentium II. szerkezete hasonlít az ábrához. Ezt majd a 4. fejezetben tárgyaljuk. A szuperskalár architektúrát ebben a megközelítésben 1987-ben alkották meg (Agerwala és Cocke, 1987), bár gyökereiben több, mint 30 évet vezet vissza a CDC6600-as számítógéphez. A 660-as 100 nsec-enként kérdezett le egy utasítást és a 10 funkciós egység egyikéhez továbbította párhuzamos feldolgozásra, mialatt a CPU a következő utasítást kérte.

2-6. ábra: egy szuperskalár processzor 5 funkciós egységgel

Magától értetődően a szuperskalár processzorok ötlete az volt, hogy az S3-as szakasz észrevehetően gyorsabban kezeli az utasítást, mint ahogy az S4-es szakasz végrehajtja. Ha az S3-as szakasz minden 10 nsec-ben kiad egy utasítást és minden funkcionális egység 10 nsec alatt el tudja végezni a feladatát, akkor egyszerre csak egy lenne elfoglalt, érvénytelenítve az egész ötletet. Valójában a 4-es szakaszbeli funkcionális egységeknek észrevehetően több, mint egy ciklusideig tart a végrehajtás, pontosan azoknak, amelyek elérik a memóriát vagy lebegőpontos számolást

csinálnak. Mint az ábrán láthatjuk, egyszerre több ALU is lehet az S1-es szakaszban.

Processzor szintű párhuzamosság

Az igény a még gyorsabb számítógépekre kielégíthetetlennek látszik. A csillagászok szimulálni akarják, hogy mi történt a "nagy bumm" első mikromásodpercében, a közgazdászok modellezni akarják a világgazdaságot és a tinédzserek 3D-s interaktív multimédiás játékokkal akarnak játszani az interneten virtuális barátaikkal. Mialatt a CPU-k gyorsulnak, lényegében problémákba ütköznek, mint pl. a fénysebesség, amely 20cm/nsec rézdrótban vagy optikai kábelben, bármilyen okosak is az Intel mérnökei. A gyorsabb chipek több hőt termelnek, aminek az eloszlatása is probléma.
Az utasítás szintű párhuzamosság egy kicsit segít, de a pipeline és a szuperskalár műveletekkel ritkán nyernek többet, mint 5-10 faktort. Hogy 50-et, 100-at vagy még többet nyerjenek az egyetlen mód, hogy számítógépeket több CPU-val tervezzenek, így megnézzük, hogy ezek hogyan szerveződnek.

Array számítógépek

Sok probléma a fizikában és a mérnököknél tömböket tartalmaz vagy másképpen magas szintű szerkezeteket használnak. Gyakran ugyanazokat a műveleteket egy időben sok különböző adathalmazon hajtják végre. A szabályossága és szerkezete ezeket a programokat kifejezetten könnyű célponttá teszi a gyorsítás és párhuzamos végrehajtás értekében. 2 módszert használunk a nagyobb tudományos programok végrehajtására. Mialatt ez a két rendszer sok szempont szerint hasonló, az egyik egy egyszerű processzor kiterjesztése, a másik pedig egy párhuzamos felépítésű számítógép.
Az array processzorok nagy számú azonos processzort tartalmaznak, melyek ugyanazt a sor utasítást hajtják végre különböző adathalmazokon. A világ első array processzora az illinoisi ILLIAC IV volt, amit a 2-7. ábrán ábrázoltunk. (Bouknight et al., 1972). Az eredeti terv az volt, hogy építeni kell egy gépet, amely 4 kvadránst tartalmaz és mindegyiknek legyen egy 8*8-as négyzethálója processzor/memória alkotóelemekből. Kvadránsonként egy egyszerű vezérlőegység átadja az utasításokat, amelyeket a többi processzor hajt végre mindegyik a saját adatait használva a saját memóriájából. (amit az inicializációs szakasz alatt töltött le) Egy négyes faktor túlterhelése következtében eddig csak egy kvadránst építettek, de ez 50 megaflopnyi haladást jelentett. (million floating-point operations per second)

***[54-57]

Azt mondják, hogy ha az egész gép kész lesz és megvalósította az eredeti teljesítménybeli célját, (1 gigaflop) megduplázná a világ számítástechnikai kapacitását. A vektorprocesszor a programozóknak úgy tűnik, hogy nagyon hasonló az array processzorhoz. Mint egy array processzor, nagyon hatékonyan hajtja végre a műveletsorozatot az adatelemek párjain. De amíg az array processzor minden fejlettebb műveletet, egyszerű, erősen pipeline-ozott számlálóban hajtja végre. A Seymour Cray által alapított Cray Research sok vektorprocesszort gyártott, kezdve a Cray-1-től 1974-ben és folytatva a jelenlegi modellekkel. (A Cray Research ma az SGI része). Mind az array , mind a vektorprocesszorok vektortömbökkel dolgoznak. Mindkettő egyszerű utasításokat hajt végre, például két vektorként az elemeket párosával egymáshoz adja. De míg az array processzor ezt úgy csinálja, hogy annyi számlálója van mint ahány eleme a vektornak, addig a vektorprocesszornál megjelenik a vektorregiszter fogalma, amely egy készlet hagyományos regisztert tartalmaz, amit egyszerű utasításként be tud tölteni a memóriába, amelyet ténylegesen sorba tölt be a memóriába. Azután egy fejlettebb vektorutasítás végrehajtja két ilyen vektor páros összeadását úgy, hogy azokat egy pipeline-olt számlálóba adagolja a két vektorregiszterből. Az eredmény a számlálóból egy másik vektor, amelyet vagy eltárol egy vektorregiszterbe, vagy egyenesen egy operandusként használ fel egy másik vektorművelethez.
Az array processzorokat még mindig gyártják, de folyamatosan csökken irántuk a kereslet, mivel csak azokon a problémákon működnek jól, amelyek ugyanazt a számolást igénylik egyidejűleg sok adathalmazon. Az array processzorok végre tudnak hajtani néhány adatkezelést, amivel eredményesebbek, mint a vektorszámítógépek, de ezek sokkal több hardwaret igényelnek és közismerten nehéz őket programozni. Ugyanakkor a vektorprocesszorokat hozzá lehet kapcsolni a hagyományos processzorokhoz. Az eredmény az, hogy a program vektorizálható részeit gyorsan végre lehet hajtani, a vektoregység előnyeit kihasználva, amíg a program többi részét végrehajtja az egyszerű processzor.

Multiprocesszorok

A feldolgozó alapelemek egy array processzorban nem független CPU-k, mivel csak egy irányítóegység osztozik rajtuk. Az első párhuzamos rendszerünk egy sokrészes processzorral a multiprocesszor. A rendszer, amely több, mint egy CPU-val osztozik a közös memórián, ahogy egy csoport ember osztozik egy teremben egy táblán. Mivel mindegyik CPU írni és olvasni tudja a memória bármely részét, összhangba kell hozni őket (a szoftverben), hogy elkerüljék az egymás útjába való kerülést. Megszámlálhatatlan megvalósítási forma lehetséges. A legegyszerűbb egy egyszerű busz sokrészes CPU-val és egy memóriával csatlakoztatva. Egy ilyen busz alapú multiprocesszor látható a 2-8.(a) ábrán. Sok cég csinál ilyen rendszereket.

2-8. (a): egy egyszerű buszos multiprocesszor.(b):egy multikompjúter helyi memóriákkal

Nem kell hozzá sok képzelőerő, hogy rájöjjünk, ha nagyszámú gyors processzor ugyanazon a buszon keresztül próbálja elérni a memóriát, az konfliktushoz vezet. A multiprocesszor tervezők sok ötlettel jöttek elő, hogy csökkentsék ezt a versengést és

növeljék a teljesítményt. A 2-8. (b) ábrán látható egy terv, ahol mindegyik processzornak ad egy saját helyi memóriát, ami nem elérhető a többi számára. Ezt a memóriát a programkódnak azon adott egységei használják, amelyhez nincs szükség osztozkodásra. Ezen memóriák elérésére nem használják a fő buszt nagy mértékben csökkentve ezzel a busz forgalmát. Más minták (pl. cachelés) is lehetségesek.
A multiprocesszoroknak meg van az az előnyük a többi fajta párhuzamos számítógépekkel szemben, hogy könnyű dolgozni az egyszerű megosztott memória programozási modelljében. Képzeljünk el például egy programot, ami cellákat keres egy mikroszkópon át lefényképezett zsebkendőn. A digitalizált fényképet a közös memóriába tárolhatjuk úgy, hogy minden processzornak kijelöljük a fotó egy részét, hogy abban keressen. Mivel mindegyik processzor elérheti az egész memóriát nem probléma olyan cella tanulmányozása, amelyik az egyik területen kezdődik, de átlóg a másikba.

Multiszámítógépek

Bár a mikroprocesszorokat kis számú processzorra (<= 64) viszonylag könnyű építeni, nagyokat meglepően nehéz megalkotni. A nehézség az, hogy az összes processzort hozzá kell csatlakoztatni a memóriához. Hogy elkerüljék ezt a problémát sok tervező egyszerűen elvetette a közös memória ötletét, és csak olyan rendszereket épített, nagy számú összekapcsolt számítógépekből, hogy mindegyiknek volt saját memóriája, de nem volt közös memória. Ezeket a rendszereket hívják multiszámítógépeknek.
A CPU-k egy multiszámítógépekben úgy kommunikálnak egymással, hogy e-mail-szerű üzeneteket küldenek egymásnak csak sokkal gyorsabban. Az olyan rendszerek, amelyben minden számítógép össze van kapcsolva, nem praktikusak, így olyan topológiákat, mint pl. 2D-s és 3D-s hálót, fát, gyűrűt használják. Ennek eredményeként az üzenetnek az egyiktől a másikig át kell haladnia egy vagy több köztes számítógépen vagy kapcsolón, hogy eljusson a forrástól a célig. Mindazonáltal az üzenetátadási idő mikroszekundumos nagyságrendben nagyobb nehézségek nélkül megvalósítható. Multiszámítógépeket közel 10000 CPU-val építettek és helyeztek üzembe.
Mivel a multiprocesszorokat könnyű programozni és a multiszámítógépet könnyebb építeni, ezért folynak a kutatások a hibridrendszerek tervezésére, amelyek ötvözik mindkét előnyös tulajdonságot. Ezek a számítógépek megpróbálják a közös memória illúzióját létrehozni anélkül, hogy a tényleges kiépítésnek költségei lennének. A multiprocesszorokat és multiszámítógépeket részletesebben a 8. fejezetben tárgyaljuk.

Elsődleges memória

A memória az a része a számítógépnek, ahol a programok és az adatok tárolódnak. Néhány számítógéptudós (főleg a britek) inkább a raktár szót használják a memória helyett. bár a raktár szót egyre többet használják a lemezkapacitásra. A processzorok által írható és olvasható memória nélkül nem lennének digitális számítógépek.

A bit

A memória alapegysége a bináris számjegy, amit bitnek hívunk. A bit értéke 0 vagy 1 lehet. Ez a lehető legegyszerűbb alapegység. (Egy csak nullákat tároló készülék nehezen lehetne egy memóriarendszer alapja, legalább két érték szükséges.)

Az emberek gyakran mondják azt, hogy a számítógépek azért használnak kettes számrendszert, mert hatékony. Amit mondanak (bár csak ritkán jönnek rá) az, hogy a digitális információt megkülönböztetve tárolhatják egy folytonos fizikai mennyiség értékei között, mint pl. a feszültség vagy az áram. Minél több értéket kell megkülönböztetni, annál kisebb a különbség két szomszédos érték között és annál megbízhatatlanabb a memória. A kettes számrendszerben csak két értéket kell megkülönböztetni. Logikusan így ez a legmegbízhatóbb módszer a digitális információ kódolására. Ha nem vagy gyakorlott a kettes számrendszerben, akkor nézd meg az A függeléket.

Néhány számítógépet, mint pl. a nagy IBM főrendszerét, tizes és kettes számrendszerrel is hirdették. Ezt a trükköt úgy vitték véghez, hogy 4 bitet használtak egy tizes számrendszerbeli szám tárolására, amit BCD-nek (Binary Coded Decimal) hívtak. Négy bittel 16 kombináció érhető el, használva a 10 számjegyet 0-tól 9-ig és hat kombinációt nem használva. Az 1944-es szám így néz ki tizes számrendszerben és simán kettes számrendszerben 16 bitet használva:

decimális: 0001 1001 0100 0100 bináris: 0000011110011000 .

Tizenhat bit decimális formában a számokat 0-tól 9999-ig tudja tárolni, csak 10000 kombinációt adva, amíg a 16 bites tisztán bináris szám 65536 különböző kombinációt tud tárolni. Ezért mondják azt, hogy a kettes számrendszer hatékonyabb.

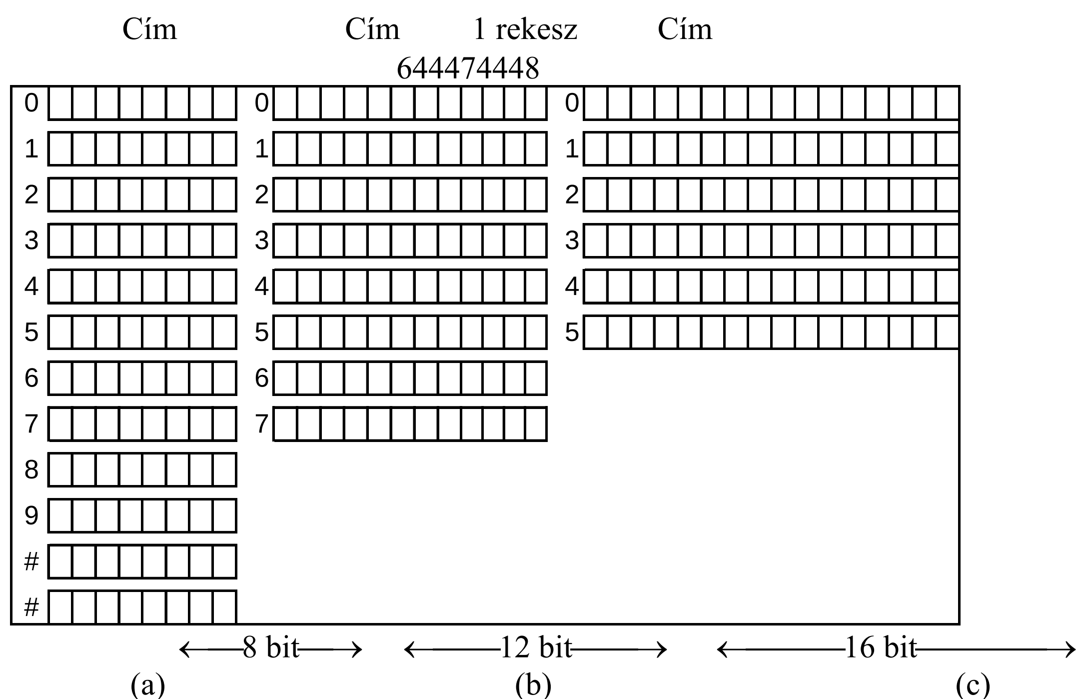
Bár gondoljuk át mi történne, ha néhány kiváló elektromérnök feltalálna egy megbízható elektronikus eszközt , ami a számjegyeket 0-tól 9-ig tudná tárolni úgy, hogy a 0 és 10 volt közötti intervallumot 10 részre osztja. Négy ilyen eszköz bármilyen számot tudna tárolni 0 és 9999 között, és így 10000 kombinációt állítana elő. Ugyanúgy használhatnánk őket bináris számok tárolására úgy, hogy csak 0-t és 1-et használna, bár így négy csak 16 kombinációt tudna tárolni. Ilyen eszközökkel a tizes számrendszer nyilvánvalóan hatékonyabb lenne.

Memóriacímek

A memóriák cellákat tartalmaznak, amelyek mindegyike tud egy kis információt tárolni. Minden cellának van egy száma, amelyet címnek hívnak, és amelyre mindegyik program tud hivatkozni. Ha egy memóriának n cellája van, akkor 0-tól n-1-ig lesz a címzése. A memóriába minden cella ugyanannyi bitet tartalmazhat. Ha egy cella k bitet tartalmaz, akkor 2^k bitkombinációt tárolhat. A 2-9-es ábra a 96 bites memória három különböző szerveződését mutatja. Vegyük észre, hogy a szomszédos celláknak egymást követő címzésük van.

A számítógépek, amelyek a kettes számrendszert használják (beleértve a nyolcas és tizenhatos számrendszer jelölését a kettes számrendszerben) bináris számként fejezi ki a memóriacímet. Ha egy címnek m bitje van akkor a megcímezhető cellák száma 2^m. Például ha egy címet nem hivatkozásra használunk a 2.9 (a) ábrán, 4 bitre van szükségünk a sorban, hogy kifejezzük az összes számot 0-tól 11-ig. Egy 3 bites cím elegendő a 2.9 (b) és (c) ábrának. A címekben a bitek száma determinálja a memóriában a közvetlenül elérhető cellák maximális számát, és független a cellánkénti bitek számától.

***[58-61]



2-9. ábra egy 96 bites memória 3-féle szervezése

Egy egyenként 8 bites 2^{12} rekeszes memóriának és egy egyenként 64 bites 2^{12} rekeszű memóriának egyenként 12 bites címekre van szüksége. A 2-10-es ábrán látható néhány, üzleti célból eladott számítógép bit/rekesz adatai.

A rekesz jelentése: a legkisebb, még címezhető egység. Az utóbbi években majdnem minden számítógépet gyártó cég a 8 bites rekeszt szabványosította, amelyet 1 **byte**-nak nevezünk. A byte-okat **szavak**ba csoportosítjuk Egy olyan számítógép esetében, melyben egy szó 32 bites 4 byte-ot számolunk szavanként; míg egy olyan gép esetében, melyben egy szó 64 bites ez az érték 8 byte szavanként. A szó tulajdonsága, hogy legtöbb utasítás az egész szóra hat, például ha két szót összeadunk. Így egy 32 bites gépnek 32 bites regiszterei vannak és olyan utasításkészlete, mely a 32 bites szavakra hat; míg egy 64 bites gépnek 64 bites regiszterei lesznek és olyan utasításai átmozgatásra, hozzáadásra, kivonásra és egyéb műveletekre melyek 64 bites szavakkal operálnak.

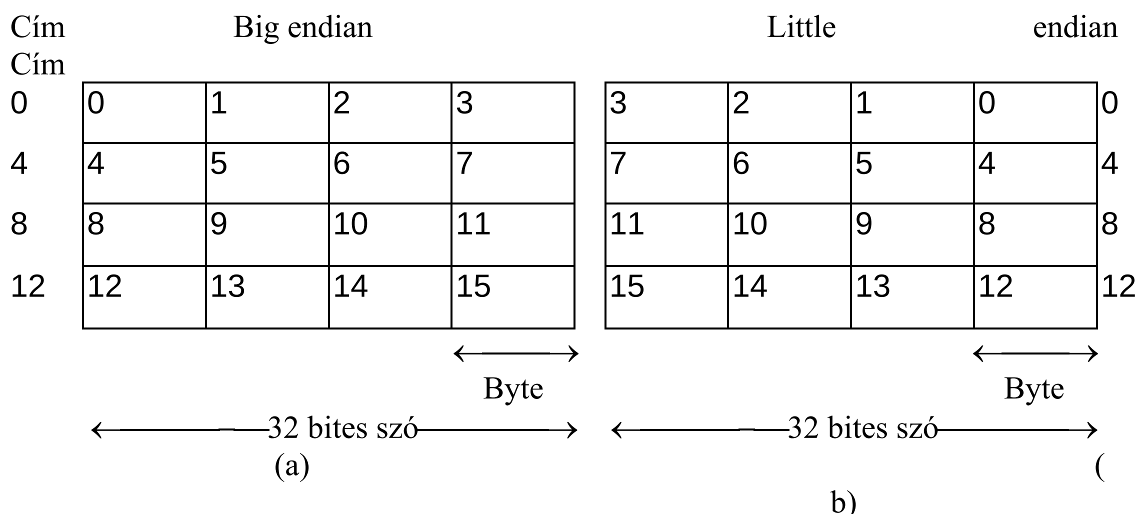
2.2.3 Byte elrendezés

A byte-ok a szavakban balról jobbra, vagy jobbról balra számozhatók. Először úgy tűnhet, hogy a választás a két elrendezés között teljesen felesleges, de nemsokára látjuk majd, hogy igenis nagy jelentősége van. A 2-11 (a) ábra egy 32 bites gép memória darabját ábrázolja, melynek byte-jai balról jobbra számozottak, úgy, mint a SPARC-nál, vagy nagy IBM mainframe gépeknél.

Computer	Bit / rekesz
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

2-10. ábra. Bit/rekesz értékek történelmünk érdekes üzleti gépeinél.

A 2-11 (b) ábra egy hasonló ábrázolását mutatja egy ugyancsak 32 bites gép memóriájának jobbról balra számozásos esetben. Ilyenek az Intel család tagjai. Az első esetben a gépet **big endian** computernek, míg a másodikat **little endiannak**. Ezek az elnevezések Jonathan Swift Gulliver utazásai nyomán születtek, melyben az író a politikusokat karikírozza azzal, hogy kemény vitákat folytatnak arról, hogy a lágy tojást a vastag végénél (big end), vagy hegyes végénél (little end) kell feltörni. A gépeknél a big endian a magasabb helyi értéktől való számozást, a little endian az alacsonyabb helyi értéktől valót jelenti. Ezt az elnevezést a számítógép architektúrában először Cohen 1981-es cikkében alkalmazta.



2-11. ábra (a) A big endian memória, (b) a little endian memória

Fontos megértenünk, hogy mindkét rendszerben egy 32 bites egész, melynek értéke, mondjuk 6 balról 29 db. 0-val jobbról 110 kettes számrendszerbeli számokkal van ábrázolva. A big endian esetében a 110 bit a 3 (vagy 7, vagy 11, stb.) byte-ban van, míg a little endian esetében ezek a 0 (vagy 4, vagy 8, stb.) byte-okban találhatók.

Mindkét esetben az egészet tartalmazó szó címe 0.

Ha a számítógépek csak egészeket tárolnának, nem lenne semmilyen probléma. Ám sok alkalmazáshoz szükség van az egészek keveredésére, karakterláncokra (string), és más adattípusokra. Gondoljunk például egy egyszerű személyi rekordra, mely egy stringet (név) és két egészet (életkor, osztály) tartalmaz. A string végén egy, vagy több 0 áll, hogy kitöltsön egy szót. Ez így nézne ki egy big endian gépen: 2-12 (a) ábra. És egy little endian gépen: 2-12 (b) ábra, ha az adatok: Jim Smith, 21 éves, 260-as (1x256+4=260) osztály.

0	J	I	M	
4	S	M	I	T
8	H	0	0	0
12	0	0	0	21
16	0	0	1	4

(a)

	M	I	J	0
T	I	M	S	4
0	0	0	H	8
0	0	0	21	12
0	0	1	4	16

(b)

	M	I	J
T	I	M	S
0	0	0	H
21	0	0	0
4	1	0	0

(c)

J	I	M		0
S	M	I	T	4
H	0	0	0	8
0	0	0	21	12
0	0	1	4	16

(d)

2-12. ábra (a) Személyi rekord egy big endian gépnél, (b) ugyanez a little endiannál
(c) a big endianból a little endianba való adatátvitel eredménye
(d) a byte cserélés eredménye (c)-ből.

Mindkét ábrázolás jó és nincs bennük belső ellentmondás sem. A problémák akkor kezdődnek, amikor az egyik gép megpróbálja elküldeni a rekordot a másiknak egy hálózaton keresztül. Tegyük fel, hogy egy big endian rendszer küldi a rekordot egy little endian rendszernek, időegységenként 1 byte-ot, kezdve a 0. byte-tól a 19. byte-tal bezárólag. (Optimisták vagyunk, és feltesszük, hogy a byte bitjei átvitel közben nem fordulnak meg). Így a big endian gép 0. byte-ja a little endian gép 0. byte-jára kerül és így tovább. Ez látható 2-12 (c) ábrán is.

Amikor a little endian megpróbálja kinyomtatni a nevet, sikerül neki, de az életkor $21x2^{24}$ lesz, és az osztályszámot is megcsonkítja. Ez a jelenség azért áll elő, mert az átvitel során egy szóban a karakterek sorrendje megfordul, ahogy kell, de az egész számban is felcserélődnek a byte-ok, aminek nem szabadna megtörténnie.

Egy nyilvánvaló megoldás az lenne, hogy a software-el megfordíttatjuk a byte-okat aztán, hogy a másolás megtörtént. Ekkor a 2-12 (d) ábrán látható elrendezést kapjuk, melyben a két egész helyes lesz, viszont a karakterlánc "MIJTIMS" lesz egy "H"-val a végén, mely sehová sem tartozik. Ez a felcserélődés azért jön létre, mert olvasás közben a gép először a 0. byte-ot olvassa, utána az 1.-t, és így tovább.

Valójában nincsen egyszerű megoldás. Egy lehetőség, amely működik, de nem hatékony az, hogy egy fejet iktatunk be minden adat elejére, amely megmondja, hogy milyen adattípus követi (string, egész, vagy más) és milyen hosszú. Így a fogadó gép csak a szükséges konverziókat hajtja végre. Bárhogyan is, de tisztában kell lennünk azzal, hogy a byte elrendezésben a szabványok hiánya jelentős kellemetlenségeket okozhat adatátvitel közben különböző gépeknél.

2.2.4 Hibajavító kódok

A számítógépek memóriái néha hibákat vétenek a tápkábel feszültségingadozása, vagy egyéb okok miatt. A hibák elleni védekezésül néhány memóriafajta hibakereső, vagy hibajavító kódokat használ. Amikor ezeket a kódokat

alkalmazza a memória, újabb biteket rak minden memória-szóhoz egy speciális módon. Amikor olvasásra kerül sor, ezek a bitek ellenőrzésre kerülnek, s a memória így veszi észre egyes hibák előfordulását

Hogy megértsük, hogyan is kezeli a memória a hibákat, fontos, hogy először azt értsük meg, hogy mik is valójában a hibák. Tegyük fel, hogy egy memória-szó áll m adatbitből, amihez hozzáadunk r redundáns, vagy ellenőrző bitet. Legyen az egész hossz n (n=m+r). Egy n bites egységet, melyet m adat és r ellenőrző bit alkot n bites **kódszó**nak nevezünk.

Adjunk meg két kódszót, mondjuk: 10001001 és 10110001. Meghatározhatjuk, hogy a megfelelő bitek közül hány különbözik. Estünkben ez szám 3. Hogy meghatározzuk, hogy hány bit különbözik, alkalmazzuk a két kódszóra a bitenkénti *kizáró vagy* műveletet (XOR) és számoljuk meg az 1-esek számát az eredményünkben! Azt a számot, amellyel két kódszó eltér egymástól **Hamming-távolságnak** is nevezik (Hamming, 1950). Tulajdonsága az, hogy ha két kódszó Hamming-távolsága d, akkor éppen d egyes hibára van szükség ahhoz, hogy az egyiket a másikba vigye. Például, ha a két kódszó 11110001 és 00110000, akkor Hamming-távolságuk éppen 3, így 3 egyes hibának kell történnie, hogy a két kódszót egymásba vigye.

Egy m bites memória-szóban az összes 2^m bit-elrendezés lehetséges, de az ellenőrző bitek számítási módja miatt a 2^n bit-elrendezésből csak 2^m bit-elrendezés érvényes. Ha a memória olvasás közben érvénytelen kódszót talál, gép tudni fogja, hogy memóriahiba történt. Ismerve az ellenőrző bitek számítási algoritmusát, készíthetünk egy teljes listát az érvényes kódszavakról, és erről a listáról kikereshetjük azt a két kódszót, melyek Hamming-távolsága a legkisebb. Ez a távolság lesz az egész kód Hamming-távolsága.

A hibakeresés és hibajavítás tulajdonságai függnek a kód Hamming-távolságától. Ahhoz, hogy a gép d darab egyes hibát megkeressen d+1 távolságú kódra van szüksége, mert egy ilyen kóddal d darab egyes hiba képtelen egy érvényes kódszót egy másik érvényes kódszóra változtatni. Hasonlóan, ahhoz, hogy a gép d darab egyes hibát kijavítson 2d+1 távolságú kódra van szükség, mert így a lehetséges kódszavak olyan távolságra vannak egymástól, hogy még d bitnyi változtatás esetében is, az eredeti kódszóhoz még mindig közelebb vannak, mint a többihez, így ez egyedülálló módon meghatározható.

Egy egyszerű példaként a hibakereső kódokhoz vegyünk egy olyan kódot, amelyben egy egyszerű paritás-bitet csatolunk az adathoz. A paritás-bit megválasztott, így az 1-es bitek száma a kódszóban páros (vagy páratlan). Egy ilyen kódnak 2 a távolsága, mivel bármely egyes bit hiba olyan kódszót generál, melynek rossza paritása. Tehát, ez a módszer csak egyes bithibák észlelésére alkalmazható.

***[62-65]

ez két egybites hibát rendel hozzá egy ismert kódszóhoz egy másik kódszót.
Ezt használhatjuk arra hogy megtaláljunk egyszeru hibákat.Amikor a szó a rossz paritást tartalmazza, amit a memoriából olvas be, a hibafeltétel teljesül.
A program nem tud folytatódni, de legalább nincs helytelen számitás.
Egy egyszeru példa a hiba-javitás kódra,amely négy ismert kódszót tartalmaz:

0000000000, 0000011111, 1111100000, és a 1111111111

Ennek a kódnak 5 távalsága van, ami azt jelenti hogy dupla-hibákat is ki tud javitani.
Ha 0000000111 kódszó érkezik, a kapó tudja, hogy az eredeti kód a 0000011111 volt(ha nem volt dupla-hibánál több hiba).
Ha valahogy egy tripla-hiba a 0000000000 - t kicseréli 0000000111 - re, a hiba nem javitható ki.
Képzeld el ,hogy el akarunk tervezni egy kódot m adat-bit-tel és r vizsgáló-bittel, ami lehetové teszi minden egybites hiba kijavitását.Mindegyik kódszó a 2m ismert kódszó közül van n illegális kódszó az elso szinten.Ezek át vannak formálva szisztematikus invertálással, vagyis minden n bithez tartozik egy n-bites kódszó.
Igy minden 2m ismert kódszóhoz szükséges n+1 bites minta dedikálása(az n lehetséges hiba és a javitó minta).
Mikor a bit-minták teljes száma 2n, akkor (n+1)2m<2n. Felhasználva n=m + r kapjuk ,hogy (m + r + 1)<2r. Adott m, ez egy kisebb határt szab a tesztelobitek számát illetoen , amik az egyszerü hibákat javitják.
A 2-13 számok megmutatják hogy mennyi tesztelobit szükséges a változó szóméretekhez.

szó méret	tesztelo bitek	totál méret	A fentiek aránya
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

2-13.
A tesztelobitek száma abban a kódban amely az egyszeru hibákat javitja.

Ez az ideálisan kisebb határ megvalósitható felhasználva Richard Hamming módszerét(1950).Mielott megnéznénk Hamming algoritmusát, nézzünk meg egy egyszeru grafikus reprezentációt , ami jól illusztrálja a 4-bites szavak hibajavitó kódját.A 2-14(a). venn diagramja tartalmaz három halmazt, A,B és C, melyek együt 7 halmazrészt alkotnak.
Példaként vegyük egy 4-bites memóriaszó a 1100 kódolását az AB,ABC,AC és BC részhalmazokban,

1 bit egy részhalmazonként(ABC sorrendben).
Ezt a kódolást mutatja a 2-14(a). kép.

Azután mind a 3 üres halmazhoz hozzáadunk egy páros bitet, azért hogy
a párosságot biztositsuk, ahogy ez a 2-14(b) ábrán is látható.
Definiálva, a három halmazban(A,B,C) a bitek összege mostmár páros szám.
Az A halmazban 4 szám van:0,0,1 és 1, amelyeket ha összeadunk 2 lesz belole,
ami páros szám.
A B halmazban a számok:1,1,0 és 0, amelyeket összeadva szintén 2-ot kapunk,
amely páros szám. Végül a C halmzban ugyanezt kapjuk.
Ennél a példánál az összes halmaz ugyanaz, de a 0 és a 4 összege más példákban is
lehetséges.
Ez az ábra egy jeligével egyezik meg:4 databit és 3 páros bit.
Most tegyük fel hogy a bit az AC halmazban tönkre megy, a 0 - át 1 - re változtatva,
ahogy a 2-14(c) ábrán is látható. A számitógép láthatja hogyaz A és C halmazokban
rossz(páratlan) számok vannak. Az egyetlen módja annak, hogy ezt kijavitsuk (single-bit csere)
az hogy az AC halmazt visszahelyezzük 0 - ra, és igy javitjuk ezt a hibát.
Ily módon a számitógép automatikusan a single-bit memóriát.
Most nézzük meg hogyan használható a Hamming féle algoritmus arra hogy létrehozzunk hibajavitó
kódokat bármilyen méretu memória szóra.A Hamming kódban, r páros bitek az m-bites
szóhoz vannak hozzáadva, igy m+r bit hosszuságú új szót létrehozva ezzel.
A bitek meg vannak számozva, 1 - gyel és nem 0-val kezdodve, a bit 1- gyel a
legbaloldalibb bit. Minden bit melynek bitszáma 2 hatványa,ezeket data-ra használják
fel.
Például:egy 16-bites szóval, 5 páros bitet adnak hozzá.
1,2,4,8 és 16 az páros bit, és az összes maradék data bit.
Egyben, a memória szónak 21 bitje van(16 data és 5 páros bit).
Páros számot fogunk használni ebben a példában.Minden egyes páros bit
bizonyos bit helyet foglal el: a páros bit adott úgy hogy az 1s teljes
száma az elfoglalt helyen páros. A páros bitek által elfoglalt helyzetek.

Bit 1 teszteli 1,3,5,7,9,11,13,15,17,19,21 biteket

Bit 2 teszteli 2,3,6,7,10,11,14,15,18,19 biteket

Bit 4 teszteli 4,5,6,7,12,13,14,15,20,21 biteket

Bit 8 teszteli 8,9,10,11,12,13,14,15 biteket

Bit 16 teszteli 16,17,18,19,20,21 biteket.

Általában a b bit azok a bitek által van elfoglalva b1,b2,..,bj olyan
mint b1+b2+...+bj=b. Például,5-ös bit 1 és 4 bittel helyettesitheto, mert
1+4=5. A 6-os bit 2 és 4-vel helyettesitheto mert 2+4=6 etc..
2-15. ábra egy Hamming féle kód felépitését mutatja:16bit memória szó

1111000010101110. A 21bites kód a 001011100000101101110.
Hogy megnézzük hogy muködik a hibajavitás, vegyük figyelembe mi történne ha az 5-ös bit reciprokát vennénk.a hatványkitevoben. Az új kódszó a 001001100000101101110 lenne a 001011100000101101110 helyett.
Az 5 páros bit helyettesitve lesz a köv eredménnyel:

Páros bit 1 téves (1,3,5,7,9,11,13,15,17,19,21 tartalmaz öt 1s-t)

Páros bit 2 helyes (2,3,6,7,10,11,14,15,18,19 tartalmaz hat 1s-t)

Páros bit 4 téves (4,5,6,7,12,13,14,15,20,21 tartalmaz öt 1s-t)

Páros bit 8 helyes (8,9,10,11,12,13,14,15 tartalmaz két 1s-t)

Páros bit 16 helyes (16,17,18,19,20,21 tartalmaz négy 1s-t)

Az 1s teljes száma az 1,3,5,..21 bitekben páros számnak kell lennie mert páros számokat használunk.A helytelen bit párossággal helyettesitett bitek egyike kell hogy legyen bit1, nevezetesen bit 1,3,5,7,..,21.
Páros 4bit az helytelen, azt jelenti hogy a bitek egyike 4,5,6,7,12,13,14,15,20 vagy 21 helytelen.A hiba a bitek egyike kell hogy legyen mindkét listában/számsorban
,nevezetesen 5,7,13,15 vagy 21.Azonban, 2bit helyes kihagyva 7 és 15- ötöt.
Hasonlóan bit 8 helyes kihagyva 13 -at.Végül, bit 16 az helyes, kihagyva 21-et.
Az egyetlen megmaradt bit az bit 5, amely az egyetlen hibás.Mivel 1-nek volt olvasva,
0 kell hogy legyen. Ilyen módon,a hibákat ki lehet javitani.
A helytelen bit megtalálására egyszeru módszer létezik, mégpedig:
ki kell számitani az összes páros bitet. Ha mindegyik helyes, akkor nem volt hiba(vagy több mint 1). Azután össze kell adni az összes helytelen páros bitet, 1-et számolva bitnek, 1,2 bit 2-nek, 2,4 bit4 -nek etc..
A végeredmény a helytelen bit helye.
Például: ha a páros bitek: 1 és 4 helytelen, de 2,8 és 16 helyes, bit5(1+4) volt megforditva.

2.2.5 Cache Memória

Történelmi szempontból, CPU-k mindig gyorsabbak voltak mint a memóriák.
Ahogy a memóriák fejlodtek, úgy a CPU-k is, megorizve a kiegyensúlyozatlanságot.
Valójában ahogy lehetové válik hogy egyre több és több áramkört tegyünk egy chipre,CPU tervezok csovezetékekhez és szuperskalár muveletekhez használják ezeket az új lehetoségeket, és igy még gyorsabbá teszik a CPU-t.
A memóriatervezok általában új technológiát használnak hogy növeljék a chipek kapacitását,nem a sebességet, igy a probléma rosszabbnak látszik egy ido múlva. Amit a kiegyensúlyozatlanság jelent a gyakorlatban az az hogy miután a CPU kiad egy memória parancsot,nem fogja megkapni a szót amire szüksége van sok CPU szakaszra.
Minél lassab a memória,annál több szakaszt kell a CPU-nak várni.
Amint azt már kiemeltük az elobb,két módja van a probléma megoldásnak.

A legegyszerubb mód az hogy elkezdjük a READ memóriát amikor találkoznak
de folytatja a végrehajtást és lassitani a CPU-t, ha egy instrukció
megpróbálja hsználnia memória szót mielott az megérkezik.
Minél lassabb a memória annál többször fog ez a probléma elofordulni,
és annál nagyobb a büntetés amikor ez elofordul.
Például: ha a memória késleltetés az 10 kör, nagyon valószínu, hogy
a következo 10 instrukció megpróbálja a READ szót használni.
A másik megoldás az, olyan gépeket használni, amelyek nem lassitanak
de helyette a forditoprogram úgy fejlessze a kódot hogy használja a szavakat
mielott azok megérkeznek.
A baj az hogy ezt a feladatot könnyebb elmondani mint megtenni.
Gyakran a LOAD után nincs semmi más tennivaló, igy a programot kényszerítjük
hogy helyezze be a NOP(no operation) instrukciót,amely semmi mást nem tesz
mint idot pazarol.Tulajdonképpen, a probléma nem a technológia, hanem
a gazdaság.
A mérnökök tudják hogy kell megépíteni a memóriákat amelyek gyorsak mint
a CPU-k,de hogy teljes sebességen futtassák, el kell helyezniük a CPU
chipjén(mert a memóriába jutás nagyon lassú). Nagy memóriát rakva a CPU-ra
azt nagyobbá teszi, ami miatt drágább lesz, és még ha az árak nem is számítanának,
akkor is meg van szabva mekkora lehet egy CPU chip.
És itt jön ben az hogy vagy egy kicsi gyors memóriájú, vagy egy nagy
lassú memóriájú chipet vegyünk. Amit mi szeretnénk az egy nagy , gyors
memóriájú és olcsó.
Elég érdekes hogy a technikákat/módszereket úgy ismerik hogy kicsi, gyors
memóriájút egy nagy lassú memóriájút kombinálnak hogy megkapják a gyors
memóriájú sebességét és a nagy memóriájú kapacitását elfogadható áron.
A lassú, gyors memóriát cache -nek hivják(azt jelenti hogy elrejteni)
Lent röviden leirjuk hogy a Cacheket hogy használják, és hogy muködnek.
Az alapötletaz egyszeru: a leggyakrabban használt memóriaszavak a cacheben
vannak tárolva. Amikor a CPU-nak szüksége van egy szóra, eloször a cacheben
nézi meg. Csak ha a szó nincsen ott akkor megy a fo memóriába.
Ha a szavak egy alapveto részea cache-ben van, az átlagos hozzáférhetoségi
idot le lehet csökkenteni.

***[66-69]

Nem készült el: Szabó Zoltán: h938341

***[70-73]

70-73. oldalak

A szalagok és az optikai lemezek általában nincsenek szabályozva, tehát a kapacitásukat csak a tulajdonos pénztárcája határozza meg.
Harmadszorra, az elköltött pénzért kapott bitek száma csökkenti a hierarchiát. Jóllehet az árak gyorsan változnak, a központi memória dollárokban (egy megabájtért) mérhető, a mágneslemezes tároló pennykben (egy megabájtért), és mágnesszalag pedig dollárokban egy (gigabájtért) vagy annyiban sem.
Eddig már beszéltünk a regiszterről, cache-ről és a központi memóriáról. A következő részekben beszélni fogunk a mágneslemezekről, azután pedig az optikai lemezekről. A szalagokkal nem foglalkozunk, mert alig használjuk őket, és sokat nem is lehetne mondani róluk.

2.3.2 Mágneslemezek

A mágneslemez tartalmaz egy vagy több mágnesbevonatú alumínium korongot. Eredetileg ezeknek az átmérője 50 cm körül volt, de mostanság már csak 3-tól 12 cm-ig terjednek, de a notebook számítógépek lemezei már 3 cm alatt vannak, és még nincs vége. Az indukciós tekercses lemezfej a felszín felett "lebeg", csak egy vékony réteg levegő választja el a felszíntől (kivéve a floppy lemezt, ahol a fej hozzáér a felülethez). Amikor pozitív vagy negatív áramlatok mennek át a fejen, azok magnetizálják azt éppen csak a felület alatt, ezzel felsorakoztatva a mágneses részecskéket balra vagy jobbra, attól függően, hogy merről vonzza. Amikor a fej áthalad egy mágnese tér felett, akkor a pozitív vagy negatív áram begerjed a fejben, lehetővé téve azt, hogy az előzőleg tárolt adatokat visszaolvassa. Tehát amint a tálca forog a fej alatt, adatok áradatát lehet írni, és később visszaolvasni. A 2.19-es ábrán a lemez sávjának felépítése látható.
A teljes körforgás alatt megírt adatok (bitek) sorozata egy sáv. Minden sáv fel van osztva meghatározott hosszúságú szektorra, általában 512 bájtot tartalmaz, azonban ezt megelőzi a bevezető rész, amely lehetővé teszi, hogy a fej szinkronba álljon a leolvasás vagy az írás előtt. Az adatok után következik az úgynevezett Hibaelhárító kód (ECC = Error-Correcting Code) (mindkettő Hamming kód) vagy a Reed-Solomon kód, ami több hibát ki tud javítani egyszerre. Az egymás utáni szektorok után következik az átszelő rés. Néhány gyártó átalakítatlan formában hivatkozik termékük kapacitására, de a becsületesebbek az átalakított formában, amiben nincs beleszámolva a bevezető rész, az ECC és a rés mint adat. Az átalakított. Az átalakított kapacitása általában 15 százalékkal alacsonyabb, mint az átalakítatlané.
Minden lemeznek van mozgatható karja, ami képes kifelé és befelé mozogni a tengelyből más-más sugárirányú távolságban, amit a korong mozgat. Minden forgás után egy újabb programot lehet írni. Tehát a sáv: a tengely körül elhelyezkedő koncentrikus kör. Egy sáv szélessége attól függ, hogy mekkora a fej, és hogy milyen pontosan lehet azt sugárirányba elhelyezni. A mai technológiával 800-2000 sáv lehetséges centiméterenként, és a sáv szélessége 5-10 mikron között. A sáv nem egy érinthető barázda, hanem csak egy sima magnetizált anyagú gyűrű, apró védő területekkel körülötte, ami elválasztja a többi sávtól.
A hosszirányú adatsűrűség a sáv körül különbözik a sugárirányútól. Mindez sokban függ attól, hogy milyen minőségű a levegő és mennyire tiszta a felszín. Jelenleg

előállított lemezek elérik az 50000-100000 bit/cm sűrűséget. Tehát egy bit kb. 50-szer nagyobb sugárirányban, mint a kerületben. Hogy a magas minőséget megtartsák, a lemezeket gyárilag szigetelik, hogy a por és a kosz ne kerülhessen a lemezbe. Ezek a meghajtók, amiről a közelmúltban beszéltük a winchester lemezek. Az első ilyen meghajtóknak (először az IBM-től) volt 30 MB zárt, rögzített tárolója, és 30 MB levehető (eltávolítható) tárolója. Valószínűleg ezek a 30-30-as lemezek emlékeztették az embereket a 30-30-as Winchester puskákra, amik olyan nagy szerepet játszottak az amerikai határvidék nyitásában, és a "winchester" név ráragadt a lemezekre.
Sok lemez tartalmaz egymás fölé rakott korongokat, mint ahogy azt láthatjuk a 2-20-as számú ábrán. Mindegyik felületnek megvan a saját karja és feje. Az összes kar egybe van kombinálva, és egyszerre tudnak mozogni más sugárirányú pozíciókban. A cylinder (cilinder) egy megadott sugárirányban lévő sávok összessége.
Sok minden befolyásolhatja a lemez teljesítményét. Ahhoz, hogy írjon vagy olvasson egy szektort először a kart a jó sugárirányba kell mozdítani. Ez a keresés. Az átlagos keresési idő 5-15 mikro-szekundum alatt van. Amikor a fej sugárirányban van beállítva, van egy kis szünet, amit rotációs lappangásnak hívunk, amíg a kívánt szektor beáll a fej alá. A legtöbb lemez 3600, 5400 vagy 7200 fordulatot tesz meg percenként, és így az átlag szünetidő 4-8 mikro-szekundum. 10800-on forgó lemezek is elérhetőek. Az átviteli idő függ a hosszirányú sűrűségtől és a forgási sebességtől. Az átlagos átviteli arány, ami 5-től 20 MB / másodperc és az 512 bájtos szektor 25 és 100 nano-szekundum között van. Ezt követően a keresési időtől és a rotációs lappangástól függ az átviteli idő. A szektorok összevissza keresése a lemezen meglehetősen gazdaságtalan kezelésre utal.
Érdemes megemlíteni, hogy nagyon nagy különbség van egy meghajtó maximális gyorsasági aránya (burst rate) és a maximális fenntartó aránya (sustained rate) között. A maximális burst rate az az adatarány, amikor a fej túlhalad az első adatbiten. A számítógépnek az adatokat nagyon gyorsan kell kezelnie. Igaz, hogy a meghajtó azt az arányt csak egy szektorig tudja fenntartani. Néhány használatnál, pl.: multimédia, ami számít az az átlagos sustained rate több másodpercen keresztül, amit figyelembe kell venni a szükséges kereséseknél és a rotációs késleltetéseknél is.
Ahogy a lemezek forognak, 60-tól 120 forgás / másodpercig, úgy felforrósodnak és kitágulnak, megváltoztatva a tényleges geometriájukat. Néhány meghajtónak újra kell kalibrálni a beállító mechanizmusát időnként, hogy kompenzálódjon a táguláshoz. Ezt úgy teszik, hogy teljesen ki, vagy teljesen beerőltetik a fejet. Ilyen újrakalibrálások pusztítást is okozhatnak a multimédiás applikációk, mert azok több-kevesebb bit egyenletes áramlatát várják a maximális sustained rate-ban. Hogy kezelni tudják a multimédiás applikációkat, néhány gyártó audio-vizuális lemezmeghajtókat készít, amikben nincs meg az az ún. termális újrakalibráció.
Egy kis gondolkodás, meg némi középiskolai matek elárulja (K=2(r), hogy a külső barázdáknak több a hosszirányú távolsága, mint a belsőnek. Minthogy az összes mágneses lemez állandó szögben és állandó sebességgel forog, mindegy, hogy a fej hol van, az a megfigyelés mindenképpen problémát jelent. A régebbi meghajtókban a lehető legtöbb hosszirányú sűrűséget használták a gyártók a legbelsőbb barázdákon, és sikeresen a legkevesebb a legkívül levőkön. Pl.: ha egy lemeznek 18 szektora volt, akkor minden egyes szektorban 20 ívfok volt, mindegy, hogy melyik cilinderben volt. Mostanság már stratégiát használnak. A cilindereket felosztják zónákra (átlagosan 10-30 meghajtóként) és minden zónában növelik a szektorok számát bentről kifelé menve. Ez a csere nehezebbé teszi az információ követését, de növeli a meghajtó kapacitását, ami fontosabb. Az összes szektor egyforma méretű. Szerencsére néhány

dolog az életben nem változik.
A meghajtóval van összekötve a lemezvezérlő, egy chip, ami irányítja a meghajtót. Néhány vezérlő tartalmaz teljes CPU-t. A vezérlő parancsokat kap a szoftvertől (olvasás, írás és lemezformázás), hibákat fedez fel és javít ki, és átalakítja a 8 egységes bájtokat a memóriából egy sorozatos egység hullámba és ugyanezt visszafelé. Néhány vezérlőnek nem jelent gondot több szektor egymásba ütközése sem, elrejtett szektorok olvasása jövőbeni használatra, és újratérképezni a rossz szektorokat. Ez az utolsó funkció hibás szektoroktól keletkezik, amiben maradandó magnetizálódás jön létre. Amikor a vezérlő felfedez egy rossy szektort, az helyettesíti azt egy tartalék szektorral, ami erre van megspórolva ugyanazon zónán vagy cilinderen belül.

2.3.3 Floppy lemezek (Hajlékonylemezek)

A személyi számítógépek megérkezésével szükség volt találni egy utat a szoftver szétterjesztésére. A megoldás a floppy lemez (diskette) volt, kicsi és törhetetlen, mert hajlékony. A floppy lemezt eredetileg az IBM találta fel a saját karbantartására az alkalmazottak részére, de a személyi számítógépgyártók elsajátították azt, mert így sokkal kényelmesebb volt a helyzet a szoftverek eladásával kapcsolatban.
Az általános tulajdonságai ugyanazok, mint azoknak a lemezeknek, amit eddig leírtunk, kivéve azt, hogy a kemény lemezeknél a fej nem ér hozzá a felülethez, de a floppynál igen. Ezért a média és a fej is viszonylag gyorsan elkopik. Hogy ezt a folyamatot, mármint a fej kopását, késleltessük, amikor a meghajtó nem ír vagy nem olvas éppen, akkor a személyi számítógép felhúzza a fejet és megállítja a forgást. Ebből adódóan, amikor a következő olvasó vagy író parancsot kapja, akkor van egy körülbelül 1/2 másodperces várakozási idő, amíg a motor felpörög a kívánt sebességre.
Két méret létezik: 5.25 és 3.5 colos. Mindkettőnél van kisebb sűrűségű (LD = Low-Density) és nagyobb sűrűségű (HD = High-Density) változata. A 3.5-ösöknél kemény borításuk van, hogy jobban védjék azt, úgyhogy ezek nem igazán floppyk. Mivel a 3.5-ös lemezek több adatot képesek tárolni és jobban védettek, ezért lecserélték az 5.25-ösöket rájuk. A négy típus legfontosabb paraméterei a 2-21-es rajzon láthatók.

2.3.4 Az IDE lemezek

A modern személyi számítógép lemeze az IBM PC XT lemezéből emelkedett ki, ami egy 10 MB-os ún. Seagate lemez volt, egy bedugható kártyán lévő Xebec lemezvezérlő által irányított lemez. A Seagate lemeznek 4 feje volt, 306 cilindere, és 17 szektora/sávonként. A vezérlő képes volt két meghajtót is ellátni. Az operációs rendszer a megfelelő paramétereket a CPU regisztereibe teszi, amikor a BIOS-t meghívja, amely a PC-k beépített ROM-jában található, így képes írni a lemezre ill. olvasni a lemezről. A BIOS kérésére a gép utasításai betöltődnek a lemezvezérlő regisztereibe és beállítják az átvitelt.

***[74-77]

Balázs Csaba
h938290@stud.u-szeged.hu

Eleinte a meghajtóvezérlőt egy különálló kártyán helyezték el, ám ez a technológia hamarosan átalakult, és vezérlőt a meghajtóval egybeintegrálták. Ez az 1980-as évek közepén az **IDE** (Integrated Drive Electonics) meghajtókkal indult. Azonban, a BIOS hívás megállapodást nem változtatták meg, a visszafelé kompatibilitás miatt. Ezek a hívás megállapodások címezték meg a szektorokat úgy, hogy megadták azok fej, cilinder és szektor számát. A fej és cilinder számozás 0-tól indult és az szektor számozás 1-től. Ez a megoldás valószínűleg egy hibának volt köszönhető, amit a BIOS programozók azon része követett el, akik 8088 assemblerben dolgoztak. Az IDE-ben 4 bit a fejet, 6 bit a szektort és 10 bit a cilindert címezte, így a legnagyobb meghajtó 16 fejet, 63 szektort és 1024 cilindert tartalmazhatott, összesen 1.032.192 szektort. Ez a legnagyobb meghajtó maximum 528 MByte tárkapacitással rendelkezett, ami valószínűleg végtelennek tűnhetett abban az időben, de természetesen ma már nem. (Ma már kifogásolható, ha egy új számítógép nem tud kezelni meghajtókat egy Terabyte felett). Hamarosan megjelentek meghajtók 528 Mbyte felett, de rossz geometriával (pl. 4 fej, 32 szektor, 2000 cilinder). Nem volt lehetősége az operációs rendszernek ezek címzésére. Következménye: a lemezvezérlők azt kezdték tettetni, hogy a geometria a BIOS korlátokon belől van, de valójában a látszólagos geometriát a valós geometriára képezték le. Végül, az IDE meghajtók átalakultak **EIDE** (Extended IDE) meghajtókká, amelyek egy másodlagos címzési táblázatot használtak, az **LBA**-t (Logical Block Addressing), ahol a szektorok számozása már 0-tól indult és maximum 224-1 -ig tartott. Ennek a táblázatnak szüksége volt egy vezérlőre, hogy az LBA címet átalakítsa fej, cilinder és szektor címmé, de a határ 528 Mbyte volt. Az EIDE meghajtó és vezérlő más fejlesztéssel is rendelkezett, például képes volt 4 meghajtót kezelni 2 helyett, gyorsabb lett az átviteli sebesség, és tudott már CD-ROM meghajtót is kezelni. IDE és EIDE lemezek eredetileg csak Intel rendszerrel működtek, mert a kapcsolódási felülete az IBM PC busz pontos másolata volt. Azonban, ma számos más számítógép is használja az alacsony áruk miatt.

2.3.5 **SCSI lemezek**

Az SCSI lemezek nem különböznek az IDE lemezektől a cilinder, sáv és szektor rendezésétől, de más a kapcsolódási felülete és magasabb átviteli sebességet tesznek lehetővé. Az SCSI története visszavezet Howard Shugarthoz, a floppy meghajtók feltalálójához, akinek a vállalkozása 1979-ben bevezette a SASI (Shugart Associates System Interface) lemezeket. Kis átalakítás és hosszú vita után, az ANSI szabványosította 1986-ban és megváltoztatta a nevét **SCSI**-re (Small Computer System Interface). Az SCSI-t "scuzzy"-nak ejtik. 1994-ben, az ANSI kiadta a korszerűsített szabványt, az SCSI-2 -t, amely jórészt lecserélte az eredeti SCSI-1 -et. A munka napjainkban is folyik az SCSI-3 -on, habár a szabványosítás már befejeződött. Néhány gyártó már teljesítette a követelményeket, hogy kialakítsák az SCSI-3 -at. Az SCSI meghajtók magas átvitelt tesznek lehetővé, ezért ezt a szabványos lemez a UNIX munkaállomásoktól kezdve a SUN, HP, SGI keresztül más rendszerig. Ez a szabvány a Macintosh gépeken és az újabb Intel PC-ken, különösen a hálózati szervereken. Néhány jobban elterjedt paraméter a 2.22. ábrán látható.

2.22. ábra:

Név	Adat bitek	Bus MHz	Mbyte/sec
SCSI-1	8	5	5
SCSI-2	8	5	5
Fast SCSI-2	8	10	10
Fast & Wide SCSI	16	10	20
Ultra SCSI	16	20	40

Az SCSI több, mint egy merevlemez, egyben kapcsolódási felület is. Ez egy busz az SCSI vezérlőhöz, amelyhez több mint 7 eszközt lehet csatlakoztatni. Ez egy vagy több SCSI merevlemezt, CD-ROM-ot, CD írót, szkennert, kazettás egységet vagy más SCSI perifériát jelent. Mindegyik eszköz rendelkezik egy azonosítóval (ID-vel) 0-tól 7-ig (15-ig wide SCSI esetén), és két kivezetéssel: egy kimenttel és egy bemenettel. Az eszköz kimenetét összekötik a következő bemenetével, így sorban, mint a karácsonyfa lámpák fűzérét. Az utolsó egységnél azonban meg kell akadályozni a visszaverődést az SCSI busz végétől. A vezérlő egység általában egy kártyán helyezkedik el a kábeleknél, habár ez nem szigorú követelmény a szabványban. A legelterjedtebb kábel 50 vezetékes a 8 bites SCSI-hez, amelyből 25 testvezeték, amelyek egytől egyig párosítva vannak a másik 25-tel, így kitűnő zajmentességet nyújt a nagysebességű működéshez. A 25 vezetékből 8 adat, 1 paritás, 9 vezérlő jelet továbbít, a maradék áramot illetve néhány vezeték fenntartott későbbi fejlesztésre. A 16 bites és 32 bites eszközöknek szükséges egy második kábel használata a további jelek továbbítására. A kábel akár néhány méter hosszú is lehet, a külső meghajtók, szkennerek, stb. csatlakoztatására. Az SCSI vezérlők és perifériák kezdeményezőként vagy céleszközként is tudnak működni. Általában, a vezérlő a kezdeményező, amely kiadja a parancsokat a meghajtónak és más perifériáknak, amelyek a céleszközök. Ezek a parancsok 16 bytos blokkok, amelyek megmondják az eszköznek, hogy mit kell elvégeznie. A parancsok és válaszok előfordulhatnak azonos fázisban, ezért különböző vezérlő jeleket használnak, hogy észleljék az azonos fázisokat, és eldöntsék a busz hozzáférhetőségét, amikor több eszköz akarja használni a buszt egy időben. Ezek a döntések fontosak, mert az SCSI engedélyezi az eszközök párhuzamos működését, amely nagymértékű fejlesztés: több folyamat aktív egy időben (pl. UNIX, Windows NT). Az IDE és EIDE egyszerre csak egy eszköz működését engedélyezi.

2.3.6 **RAID**

A CPU teljesítménye az utolsó évtizedben exponenciálisan növekedett, körülbelül 18 hónap alatt megduplázódik. Nem úgy mint a lemez teljesítmény. Az 1970-es években, átlagosan egy pozícionálási idő 50-100 msec között volt. A mostani pozícionálási idő 10 msec. A legtöbb technikai iparban (pl. autógyártás, repülés) 5-10-szeres teljesítménynövekedés következett be, de itt a számítástechnika zavart helyzetben van. Így a CPU teljesítmények és a meghajtó teljesítmények közti különbség még nagyobb, mint valaha. A párhuzamos működést többet és többet használják a CPU gyorsítására. Ugyanez a gondolat előfordult különböző embereknél, akik úgy gondolták a párhuzamos I/O feldolgozás is jó ötlet. 1988-ban egy tanulmányban, Patterson et al. javasolta 6 lemez speciális elrendezését, amely növelheti a lemez teljesítményét, a megbízhatóságát vagy mindkettőt. Ezt az ötletet hamarosan befogadta az ipar, és elindították egy új osztályú I/O eszközt, a **RAID**-et. Patterson et al. meghatározta RAID-et, mint Redundant Array of Disks (Olcsó lemezek redundáns tömbje), de az ipar az "I" betű jelentését megváltoztatta "Olcsó" helyett "Függetlenre". Mivel egy ellenfélre volt szükség (mint a RISC ellen a CISC), megjelent a **SLED** (Singe Large Expensive Disk).

Az alapötlet a RAID mögött az volt, hogy helyezzenek el a számítógép mellett egy

másik gépet tele lemezekkel, általában egy szervert, cseréljék ki a lemezvezérlőt RAID vezérlőre, másolják az adatokat RAID-re, és folytassák a megszokott működést. Más szavakkal, a RAID-nek úgy kellett tűnni az operációs rendszernek, mintha az SLED volna, sőt nagyobb teljesítményűnek és megbízhatóbbnak. Mivel az SCSI lemezek nagy teljesítményűek voltak és olcsók, képesek voltak 7 meghajtót kezelni egy vezérlővel (15 meghajtót wide SCSI-vel), ezért a legtöbb RAID természetesen SCSI-t használt. RAID SCSI vezérlőből és a másik gép tele SCSI lemezekkel úgy jelent meg az operációs rendszer felé, mintha SLED volna. Így nem is voltak szükségesek szoftver változtatások, hogy használják a RAID-et. Minden RAID rendszer megosztotta az adatokat a meghajtókon, hogy engedélyezze a párhuzamos működése. Néhány különböző táblázatot határozott meg Patterson et al. ennek végzésére, ők csak a RAID 0. szint - 5.szintet ismerték, de volt ott több kisebb szint is. A "szint" szó elnevezés helytelen, mivel nem tartalmazott hierarchiát, mindössze hat különböző egységet jelent. A RAID 0.szintet a 2.23.(a) ábra mutatja. Bemutatja a SLED látszólagos szerkezetét RAID-ből, ahol a sávokat k szektorra osztották, 0-tól k-1 szektorig a 0. sáv, k-tól 2k-1 szektorig az 1. sáv, és így tovább. A RAID 0.szint egymásra épülő sávokat jelent a meghajtókon, ahogyan azt a 2.24. ábra mutatja be 4 meghajtó esetén. Az adatok elosztását többszörös meghajtókon stripping-nek nevezzük. Például, ha a program parancsot ad, hogy olvasson be egy adatrészt a négy következő sávból kiindulva, akkor a RAID vezérlő a parancsot négy különálló parancsra bontja, a négy lemez részére egyet - egyet, és végrehajtja azokat párhuzamosan. Így párhuzamos I/O művelet zajlik, anélkül, hogy a program tudná. A RAID 0.szint dolgozik a legjobban a nagy kérésekkel. Ha kérés nagyobb, mint a meghajtó számszor a sávok száma, néhány meghajtó összetett kérést fog kapni, így amikor végeztek az első kéréssel, kezdik a következőt. A vezérlő megosztja a kérést, eljutatja a megfelelő parancsot a megfelelő lemeznek megfelelő sorrendben, és az összetett válasz a memóriában lesz megfelelő módon. A teljesítmény kiváló és a végrehajtás pontos. A RAID 0.szint dolgozik azonban a legrosszabbul, ha mindössze egy szektor adatát kéri a program. Az eredmény helyes lesz, de a párhuzamos működés nincs kihasználva. A másik hátránya ennek az szervezésnek, hogy a megbízhatósága rosszabb, mint a SLED-é. Ha a RAID négy lemezből áll, összesen 20000 óra, körülbelül darabonként 5000 óra, ezután elromlanak és az összes adat elveszhet. A SLED 20000 óra után romlik el, amely négyszer olyan megbízhatóságot jelent. A RAID 1.szintet a 2.23. (b) ábra mutatja, ami az igazi RAID. Itt minden lemez meg van duplázva, így ott 4 elsődleges és 4 másolat lemez van. Írás során minden sávot kétszer ír, először az elsődleges, majd a másolat lemezre. Olvasáskor pedig csak az egyik példányt használja, megosztva a betöltést a többi meghajtóval. Tehát az írási teljesítménye nem jobb mintha csak egy meghajtó volna, de az olvasási teljesítménye kétszer olyan jó. A hibatűrése kitűnő: ha valamelyik meghajtó elromlik, a másik példányt használja helyette. Helyreállítás során egyszerűen be kell helyezni az új meghajtót és át kell rá másolni a másik példányt.

***[78-81]

Számítógép rendszerszervezet

A 0 és az 1-es szintekkel ellentétben, melyek szektorszalagokkal dolgoznak, a 2. szintű RAID szó alapon dolgozik, vagy akár byte alapon is. Képzeljük el, hogy egyetlen virtuális disk minden egyes byte-ját 4 bites darabokra osztjuk, majd mindegyikhez Hamming kódot adunk, hogy olyan 7 bites szót alkossunk, melynek az 1,2 és 4-es bitjei hibafelderítésre szolgálnak (paritás bitek). Továbbá képzeljük el, hogy a 2-23 c ábrán látható 7 meghajtót "óramutató" és "rotációs" helyzetek szerint szinkronizáljuk, ekkor lehetővé válik, hogy leírhassuk a 7-bites hamming -kódolt szót, úgy hogy minden meghajtóra 1 bitet.

A Thinking Machine CM2-es számítógép használta ezt a sémát, 32 bites adatszavakat véve alapul, és hozzáadván a 6 paritás bitet, hogy egy 38 bites hamming szót alkosson, plusz 1 extra bitet a szó paritásának és így minden szót szétosztott 39 lemezmeghajtóra. A teljes átvitel mérhetetlen volt, mert 1 szektor idő alatt 32 szektornyi adatot tudott felírni. Egy meghajtó elvesztése nem okozott problémát, mivel 1 meghajtó elvesztése csupán 1 bitet vont el minden 39 bites beolvasott szóból, amot a hamming kód könnyedén tudott kezelni.

Másrészt ez a séma minden meghajtó rotális szinkronizálását igényli és csak akkor van értelmezve, ha elég nagy mennyiségű meghajtóval rendelkezik (még 32 bites adatmeghajtónál és 6 paritás meghajtónál a felesleg csupán 19%). Ráadásul igénybe veszi a vezérlőt mivel minden bitnyi idő után végre kell hajtani egy hamming ellenörzést.

A 3. szintű RAID egyszerűsített változata a 2. szintűnek (lsd 2-23 d ábra). Itt egyetlen paritásbitet számol minden egyes adatszóra és ír rá egy paritásmeghajtóra. Ugyan úgy mint a 2. szintű RAID -nél a meghajtókat tökéletesen szinkronizálni kell, mivel minden egyes adatszót több meghajtóra kell szétosztani.

Első látásra úgy tünhet, hogy egy paritás bit a hibát csak felfedezni tudja, de kijavítani nem. A véletlenszerű fel nem fedezett hibák esetében ez a megfigyelés igaz. Azonban a meghajtó összeomlása esetén gondoskodik egy teljes 1 bites hibajavítóról, mivel a rossz bit helyzete ismert. Ha egy meghajtó összeomlik, a vezérlő csak tetteti hogy az összes bit 0. Ha egy szó paritás hibás, a bit a halott meghajtóról csakis 1-es lehetett, úgyhogy 1-esre javítandó. Annak ellenére, hogy a 2. és a 3. szintű RAID magos adatszintet biztosít, a másodpercenként kezelhető elkülöníthető I/O kérelmek száma semmivel sem jobb, mint egy meghajtó esetében.

A 4-es és az 5-ös szintű RAID is szalagokkal működik, nem pedig külön paritás szavakkal, és nincs szüksége szinkronizált meghajtókra. A 4. szintű RAID (lsd. 2-23 e ábra) olyan mint a 0-ás szintű RAID, egy szalagról-szalagra paritással, melyet egy extra meghajtóra ír. Pl. ha mindegyik szalag k byte hosszú, minden szalagot egybeűzi EXCLUSIVE OR-ral, mely egy k byte hosszú paritás szalagot eredményez. Ha egy meghajtó összeomlik, az elveszet byte-ok visszafejthetők a paritás meghajtóból.

Ez a forma megakadályozza a meghajtó elvesztését, de gyengén teljesít kis módosításoknál. Ha egy szektor megváltozik, be kell olvasni az összes meghajtót, hogy újrakalkulálhassuk a paritást melyet aztán újra kell írni. Lehetőség szerint beolvashatja a régi felhasználói adatokat és a régi paritás adatokat és újraszámolhatja belőlük a paritást. Még ezzel az optimalizálással is egy kis módosítás két olvasást és

két írást igényel.

A nehéz töltés következményeként a paritás meghajtón torlódás alakulhat ki. Ez a torlódás felszámolható az 5-ös szintű RAID –del , a paritás bitek megzavarásával egységesen az összes meghajtón, kerek robin divat szerint, ahogyan azt a 2-23 f ábra mutatja. Mindemellett egy meghajtó összeomlás esetében a hibás meghajtó tartalmának újraépítése komplex folyamat.

2.3.7 CD-romok

Az utóbbi években a mágneslemezekkel ellentétben az optikai lemezek lettek a kaphatók. Sokkal nagyobb rögzítési sűrűséggel rendelkeztek, mint a hagyományos mágnes lemezek. Az optikai lemezeket elsősorban Tv programok rögzítésére fejlesztették ki, de sokkal hatékonyabban használhatták számítógépes tárolási eszközként. A potenciálisan hatalmas kapacitások következtében az optikai lemezek nagyszabású kísérlet tárgyai voltak és hihetetlenül gyors fejlődésen mentek keresztül.

Az első generációs optikai lemezt a holland Philips elektronikai konfomerátum találta fel. 30 cm-es átmérőjűek voltak és Laser Vision név alatt forgalmazták, de csak Japánba futott be.

1980-ban a Philips a Sonyval kifejlesztette a CD-t, mely gyorsan felváltotta a 331/3 RPM vinyl lemezt, ami zene rögzítésére szolgált. A CD közismertebb nevén (borítója alapján) Red Book pontos technikai adatait az International Standard (IS 10149) publikálta. (A nemzetközi mércéket az International Organization for Standardization adja ki, mely tagja a nemzetközi mércecsoportnak, mint pl. ANSI, DIN stb. Mind rendelkezik IS számmal). A lemez és a meghajtó specifikációk, mint nemzetközi szabvány megjelentetésének az volt a lényege, hogy lehetővé tegye különböző zenei kiadók, zenészek és különböző gyártók részére az együttműködést . Minden CD 120 mm átmérőjű és 1.2 mm vastag , egy 15 mm-es lyukkal a közepén. Az audio CD volt az első világpiaci digitális tárolási médium. 100 évig kellene kitartaniuk. Ellenőrizd le 2080-ban, hogy sikerült az első CD-k nek.

A CD-k nagyerejű vörös lézer segítségével készülnek, mely 0,8 mikron átmérőjű lyukakat éget a beborított eredeti (master) üveg lemezbe. Ebből egy öntőminta készül, kis dudorokkal a lézer égette lyukak helyében. Ebbe a mintába olvasztott polikarbonát gyantát injekcióznak, hogy kialakuljon a CD, mely az eredeti üveg lemez lyuk mintázatát tartalmazza. Ekkor egy nagyon vékony tükröződő alumínium réteg kerül a polikarbonátra, erre pedig egy védő lakkréteg, s végül pedig minta (cimke).

A mélyedéseket a polikarbonát alapban Pit-eknek (gödröknek) hívjuk, az érintetlen területeket pedig mezőknek (Land).

Visszajátszáskor egy sokkal gyengébb erejű 0,78 mikron hullámhosszú vörös lézerdióda pásztázza végig a gödröket és a mezőket. A lézer a polikarbonáttal bevont oldalon fut, így a gödrök dudorokként állnak ki a lézer felé, az amúgy sima felszínen. A gödrök magassága Ľ a sugár hullámhosszának, ezért a gödrökből visszaverődő fény fél hullámhosszal fázison kívül esik a környező felületről visszaverődő fényhez képest. Ennek eredményeként a két sugár rombolóan interferál (gyengítik egymást) és kevesebb fény jut vissza a lejátszó fénydetektorába, mint a mezőről visszaverődő fény. Így különbözteti meg a lejátszó a gödröt (pit) és a mezőt (land). Bár úgy tűnik a legegyszerűbbnek, hogy a pit–hez 0-t a land-hez 1-et rendeljünk, mégis sokkal megbízhatóbb pit/land, land/pit átmenethez 1–et, ennek hiánya esetén 0-át rndelni, így

ez a séma van használatban.

A gödrök és a mezők egyetlen folytonos spirálra vannak írva, mely középről halad kifelé és 32 mm-es utat tesz meg a CD széléig. A spirál 22,188 fordulatot tesz meg a lemez körül (kb. 600-at mm-enként). Széttekerve 5,6 km-t tenne ki. (A spirál a 2-24-es ábrán látható.)

Hogy a zene egységes mértékkel szóljon, szükséges hogy a gödrök és a mezők állandó, lineáris sebességgel haladjanak. Következésképpen a CD forgásának sebességét állandóan a csökkenteni kell, ahogy az olvasófej a CD belsejétől a külseje felé mozog. Belülről a forgás mértéke 530 RPM (rotate per minute), hogy elérje a 120cm/másodperc áramlási sebességet. Kívülről le kell hogy csökkenjen 200 RPM-re, hogy ugyan azt a lineáris sebességet tartsa. Egy állandó lineáris sebességű meghajtó eltér a mágneses lemezmeghajtótól mely állandó sebességgel működik, függetlenül attól, hogy a fej pillanatnyilag milyen helyzetben van. Az 530 RPM messze esik attól a 3600 - 7200 RPM-től mellyel a legtöbb lemez forog.

1984-ben a Philips és a Sony rájött, hogy a CD potenciális számítógépes adattároló, és kiadták a Yellow book-ot mely megteremtette a mai CD-ROM-ok (Compact Disk – Read Only Memory) szabványát. Hogy CD-Romok felvegyék a versenyt a már kialakult audió CD piaccal, a CD-Romokat ugyanakkora méretűre tervezték, mint az audió CD-ket, mechanikusan és optikusan is kompatibilisek, megegyezett a gyártási technológiájuk is. Eme döntés következményei azok a lassú és változtatható sebességűmotorok …..

***[82-85]

Amit a Yellow Book definiált, az lett a számítógépes adatok formája. Ez is javította a rendszer hibajavítási képességét. Ez egy lényeges lépés volt, mert igaz, hogy a zenekedvelők nem vették észre egy-egy bit elvesztését itt és ott, de a számítógép használók említették, hogy ez nem az igazi. A CD-ROM minden byte-ot 14 bites symbol tárolja. Észrevehettük, hogy 14 bit elég ahhoz, Hamming féle kódolással lekódoljunk egy 8 bites byte-ot 2 bit elhagyása nélkül. Valójában, ez egy erősebb tároló rendszer. A 14-8-as elrendezés az olvasás miatt van kialakítva a hardweren a table lookup által.
A következő szint, egy 42 symbolt összefogó csoport az 588 bites frame. Minden frame tartalmaz 192 adatbitet (24 byte). A maradék 996 bitet hibajavításra és ellenőrzésre használják. Mostanáig ez a tervezet egyenlő az audio CD-knél és a CD-ROM-oknál.
A szabvány 98 frame-t egy csoportba, a CD-ROM szektorba sorolt, amint azt mutatja 2-25-ös ábra. Minden CD-ROM szektor egy 16 byte-os bevezetővel kezdődik, az első 12 00FFFFFFFFFFFFFFFFFFFF00 (hexadecimal), amely arra szolgál, hogy a lejátszó felismerje a CD-ROM szektor kezdetét. A köv. három byte a szektor számát tartalmazza. Ez szükséges, hiszen egy CD-ROM-on keresni (mely egy szimpla adatspirál) sokkal nehezebb, mint egy mágneses lemezen, ami egyforma konkrét sávokból áll. A keresésnél a szoftver a meghajtóban kiszámítja, hogy kb. hova kell mennie, odaviszi a fejet, és akkor elkezdi keresni a bevezetőt, hogy jó volt e a számítása. A bevezető utolsó byte-ja a mód.

14 bites symbol

42 symbolt tartalmaz egy frame

588 bites frame mindegyike 24 adatbytot tartalmaz

Bevezető 98 frame épít fel egy szektort
ˇ

| | adat | ECC |

2. 25. ábra Logikai adatelrendezés a CD-ROM-on

A Yellow Book két fajta módot definiált. 1-es mód a 2-25-ös ábra szerinti elrendezést használja: 16 byte bevezető, 2048 adatbyte és egy 288 byte-os hibajavítási kód (nemzetközileg elfogadott Reed Solomon kód). 2-es mód kombinálja az adat és ECC (Error Correcting and Control) területet egy 2336 byte-os adatterületbe, azoknál a felhasználásoknál, ahol nem szükséges (vagy ahol az idő nem engedi, hogy elvégezze) a hibajavítás (t), mint az audio és video anyagoknál. A megbízhatóságról 3 különböző hiba-javító rendszer gondoskodik: a symbol-on belül, a frame-ban és a CD-ROM szektorban. Egyszeri bit hibákat a legalacsonyabb szinten javítanak, rövid sor, hasadék hiba javítása a frame szintjén történik, és akármilyen visszamaradó hibát a szektorszinten küszöböli ki. Az ár, amit fizetni kell ezért a megbízhatóságért, 98 db. 588 bites framet (7203 byte) tesz ki egy egyszerű 2048 byte-os adatnál így a hatásfok csak 28 %.

Egyszeres sebességű CD-ROM meghajtó 75 sector/sec sebességű, amely adatban mérve 153600 byte/sec 1-es módban, és 175200 byte/sec 2-es módban. Kétszeres sebességű meghajtónál kétszeres a gyorsaság STB. a magasabb szintű sebességeknél is. Egy átlagos audio CD 74 perces zenét tud letárolni, amely ha 1-es módot használ, a kapacitása 681 984 000 byte. Ez az alak általában 650 Mb-ot jelent, mert 1 Mb=2^{30} byte (1 048 576 byte), nem pedig 1 000 000 byte
Megjegyezzük, hogy egy 32XCd-ROM meghajtó (4 915 200b/sec) nem ér fel egy gyors SCSI-2 mágneseslemez-meghajtóval (10 Mb/sec) habár sok CD-ROM meghajtó használ SCSI interface-t (IDE CD-ROM meghajtók már léteznek). Ha te valóban kíváncsi az időbeli eltérésre, a különbséget néhány száz millisec.-ban kell érteni, de azt tisztázni kell, hogy a CD-ROM nem sorolható egy kategóriába a mágneseslemez-meghajtóval a kapacitásuk miatt.
1986-ban a Philips újra meglepetést okozott a Green Bookkal, hozzáadta a grafikát és az INTERLEAVE audio képességét, video és adat együtt a szektorban: egy lényeges arcvonása a multimédia CD-ROM-oknak
Az utolsó darabja a CD-ROM összetételének a file rendszer. Ahhoz, hogy ugyanazt a CD-ROM-ot lehessen használni a különböző számítógépeken, egy megegyezésre volt szükség a CD-ROM file rendszerekről. Hogy ez a megegyezés létrejöjjön sok számítógépes cég képviselője találkozott a Tahoe tónál High Sierras-ban California-Nevada határán és megterveztek egy file rendszert, melynek neve High Sierra. Ezt később kiterjesztették nemzetközi szabvánnyá (IS 9660). Ez három szintes: 1. szinten a file neve max. 8 karakteres, majd ezt követhette a max. 3 karakteres kiterjesztés (mint az MS-DOS file -nál). A file neve legfeljebb betűt, számot és aláhúzást tartalmazhat. A könyvtárak 8 karakter hosszúak lehetnek, de a könyvtárak neve nem tartalmazhat kiterjesztést. Az 1-es szint megköveteli, hogy minden file legyen e határon belüli, ami nem jelent problémát egyszeresen író módnál. Minden CD-ROM, mely egyeztetve van az IS 9660-as szabvány 1-es szintjével, olvasható MS-DOS-on, Apple számítógépeken, UNIX gépeken, vagy bármelyik más rendszeren. A CD-ROM készítők figyelembe vették ezt a sajátosságot, mint egy nagy plusz tényezőt.
IS 9660 2. Szintje biztosítja a nevek 32 karakteres hosszúságát, és a 3. Szintnek nincsenek határai. A Rock Ridge kiterjesztés (szeszéjes nevét egy városról kapta a Gene Wilder főszereplésével készült Blazing Saddles c. filmből) engedélyezi a nagyon hosszú neveket (UNIX-nál), UID, GID használatát, és jelképes láncolatokat, de ezek a CD-ROM-ok nem 1-es szintűek, emiatt nem olvasható minden számítógépen.
A CD-ROM –ok különösen népszerűvé váltak az ezeken kiadott játékok, mozik, enciklopédiák, atlaszok, és mindenféle ajánlott munkák által. A legtöbb kereskedelmi software most CD-ROM-on jelenik meg. A nagy kapacitás és olcsó előállítás kombinációja tette kényelmessé végtelen sok helyen való alkalmazását.

2. 3. 8 CD-Recordables (CD-írók)

Kezdetben a berendezéseknek elő kellet állítani egy master CD-ROM-ot (v. audio CD-t abból az anyagból), ez nagyon költséges volt. De mint az szokás a számítógép iparban, semmi sem marad tartósan drága. Az 1990-es években a CD-író már nem nagyobb, mint a CD lejátszó, ami mint átlagos periféria kapható bármely számítógép üzletben. Ezek az eszközök még különböztek a mágneses lemezektől, mert egyszer írhatóak, CD-ROM-ról nem lehet törölni. Gyorsan kitalálták egy helyet, mint háttértár a nagy merevlemezek helyett, és biztosítottak egyéneknek és kezdő cégeknek saját, kisfutású CD-ROM-ot, v. készítettek master-t eladásra, nagytömegű kereskedelmi CD másolási tervekhez. Ezeket a meghajtókat úgy ismerik, mint CD-R (CD-Recordables)

Fizikailag a CD-R egy 120mm-es polycarbonát üres lemezzel kezdődik, mint a (gyári) CD-ROM, kivéve azt, hogy tartalmaz egy 0. 6 mm széles sávot, amit a lézer használ írásnál. A sávnak van egy 0. 3 mm szinuszos kitérése, pontosan 22, 05 KHz-nél, hogy ellássa folyamatosan feedback-kel, így a forgás sebességét pontosan tudják ellenőrizni, beállítani, ha szükséges. CD-R úgy néz ki, mint egy normális CD-ROM, kivéve, hogy felül aranyszínű, ezüst helyett. Az arany szín a valódi arany rétegtől származik, ez tükröződik vissza az alumínium helyett. Az eltérés abból is adódik, hogy az ezüst CD-nek a fizikai préselése, míg a CD-R-nél a különböző pitek visszaverődése szimulálja a visszatükrözést. Ezzel készen is volnánk, feltéve, hogy ha hozzáadjuk a színréteget a polycarbonát és a visszatükröződő aranyréteg közé, amint azt a 2-26-os ábra is mutatja. Két fajta színréteg van használatban: a cianid, ami zöld színű és a pthalocianid, amely narancssárga színű. A vegyészek már végtelen sok állítást tettek, hogy melyik a jobb minőségű. Ezek színek hasonlóak a fényképezésnél használtakhoz, ez megmagyarázza, miért a Kodak és a Fuji a vezető gyártók a CD-R-eknél.
A kezdeti állapotban a színréteg átlátszó, és átengedi a lézerfényt, és az visszatükröződik az aranyrétegről. Írásnál a CD-R lézer magasabb energiát (8-16 mW) vesz fel. Mikor a sugár áttör a színréteg egy pontján, akkor a vegyi összetevők megváltoznak. Ez a változás a molekuláris struktúrában készít egy sötét pontot. Amikor visszaolvassa (0, 5 mW-on) a photodetector (fénydetector), különbséget lát sötét folt (ahol a színmegtört) és átlátszó terület (ahol a szín sértetlen maradt) között. Ez a különbség van úgy értelmezve, mint a pit, és a felület közötti különbség, mindig, ha visszaolvassák egy átlagos CD-ROM olvasón, v. audio cd lejátszón.
Egy nem újfajta Cd sértette volna a színes könyvek büszkeségét, így a CD-R megkapta az Orange Book-ot, melyet 1989-ben adtak ki. Ez a dokumentum definiálta a CD-R-t és egy új formát is: CD-ROM XA, amely biztosította a CD-R-ek újra és újra írását, egy kevés szektor ma, egy kevés holnap, egy kevés a következő hónapban. Az egymásra következő szektorcsoportokat (melyek meg vannak írva) CD-ROM track-eknek hívják.
Az egyik az első alkalmazható CD-R-ek közül a Kodak PhotoCD volt. Ebben a rendszerben a vásárló hozott egy tekercs exponált filmet, és az ő régi PhotoCD-jét a fotó processorhoz, és elment a PhotoCd-jével, amin rajta voltak az új képek a régiekkel együtt. Az új adagot, amelyik a negatívok beszkennelése által készült, felírják a PhotoCD-re mint egy külön CD-ROM track. Az újra és újra írás szükséges volt, mert egy üres CD-R túl drága volt ahhoz, hogy minden tekercs filmnél egy újat kapjon.
Nyomtatott címke
Védő réteg
Visszatükröző arany réteg
Szín réteg Sötét pont a színrétegben a lézer által kiégetve az írásnál
Polycarbonát alsóréteg
← Mozgásirányítás
Lencse
photodetector prizma
Égető lézer dióda
2-26. ábra CD-R lemez és lézer keresztmetszete (nem méretarányos). Egy ezüst CD-ROM hasonló felépítésű, kivéve azt, hogy nincs színréteg, és aranyréteg helyett alumíniumréteg van

Csakhogy az újra és újra írás egy új problémát hozott elő. A régebbi Orange Book minden CD-ROM-ot egyszeri VTOC (Volume Table Of Contents ~ állomány táblázat tartalma) –kal látott el a lemez elején. Ez a séma nem működött az újra és újra írásnál (azaz a multitrack-nál). Az Orange Book írói ezt úgy oldották meg, hogy mindegyik track-nak adtak saját VTOC-ot. A file-okat besorolták a VTOC-ba, beleértve az előző track néhány, v. az összes file-ját. Miután a CD-R-t betették a meghajtóba, az operációs rendszer átnézi az összes CD-ROM track-et, hogy lokalizálja a legtöbb szabad VTOC-ot, ez adja a lemez forgását. Emelet keres néhány file-t, de nem mindet az előtte lévő track-ből az érvényes VTOC-ból, mert az lehetséges, hogy azt a látszatot kelti, hogy a file törölve van. A track-ek csoportosíthatók session-okba, ami multisession-os CD-ROM-hoz vezet. Az átlagos audio CD lejátszó nem tudja kezelni a multisession-os Cd-t, amióta elvárják a szimpla VTOC-ot a lemez kezdeténél.

Minden track írható sima folyamatos rendszerben, megállás nélkül. Ennek a következménye, hogy a merevlemeznek, amelyikről az adat áramlik, elég gyorsnak kell lennie ahhoz, hogy azt időben átadja. Ha a file másolás kiterjed az egész merevlemezre, akkor a keresési idő a CD-R-re való adatfolyamot ’kiszáríthatja’, és ez Buffer kiürülést okozhat. A Buffer kiürülésének következménye, hogy te hozzájutottál egy szép, fényes (de egy kissé drága) alátéthez az italodnak, v. egy 120mm-es aranyszínű frisbee-hez. A CD-R software-k általában felkínálják azt az opciót, hogy összegyűjti az összes bemenő file-t egy szimpla, határos 650 Mb. CD-ROM-ot megszemélyesítő helyen, és csak azután írja rá a CD-R-re. De ez az eljárástípus megkettőzi az eredeti írási időt. Ehhez szükséges egy 650 Mb. szabad terület a merevlemezen, és…

***[86-89]

A CD-R lehetővé teszi magánszemélyeknek és társaságoknak, hogy egyszerűen másoljanak CD-ROM-okat és audio CD-ket, általában megsértve ezzel a kiadó szerzői jogait. Számos módszert dolgoztak ki, hogy az efféle kalózkodást megnehezítsék és hogy bonyolultabbá tegyék egy CD-ROM más szoftverrel történő olvasását. Ezen módszerek egyike az, hogy a fájlok méretét a CD-ROM-on több gigabyte-nak tüntetik fel, kudarcra ítélve ezzel minden olyan kisérletet, hogy a fájlokat másoljanak a CD-ROM-ról a winchesterre hagyományos másolószoftverekkel.
A fájlok valódi méretét vagy a kiadó szoftverébe építik be, vagy szokatlan helyre írják a CD-ROM-on. Egy másik módszer szándékosan hibás ECC-t helyezni bizonyos szektorokba, azt várva, hogy a hagyományos másolószoftverek "kijavítják" a hibákat. A CD-ROM szoftvere ellenőrzi az ECC-ket és ha azok helyesek, akkor nem hajlandó tovább futni. A sávok közötti távolság változtatása, valamint egyéb fizikai "hibák" alkalmazása ugyancsak számításba jöhet.

2.3.9 Újraírható CD-k (CD-RW)

Bár az emberek már hozzászoktak az egyszer írható adatrögzítőkhöz, mint pl. a papír vagy a film, igény van újraírható CD-ROM-ra is. Már létező technológia a **CD-RW (CD-ReWritable, újraírható CD)**, amely ugyanakkora, mint a CD-R.
A CD-RW-n azonban cianin vagy phtalocianin bevonat helyett ezüstből, indiumból, antimonból és tellúrból álló ötvözetet alkalmaznak rögzítőrétegként. Ennek az ötvözetnek két stabilis állapota van: egy kristályállapot és egy amorf állapot, amelyek különböző mértékben verik vissza a fényt.

A CD-RW meghajtó lézerének három erősségi fokozata van. A legerősebb fokozaton a lézer megolvasztja az ötvözetet, átalakítva azt kristályállapotúból a fényt gyengén visszaverő amorf állapotúba, ezzel mélyedést jelölve. A középső fokozaton az ötvözet megolvad és visszanyeri természetes kristályszerkezetét, ezzel sima területet képezve. A legalacsonyabb erősségi fokozaton a lézer a felület állapotát vizsgálja (olvasáshoz), de nem változtat az anyag szerkezetén.

A CD-RW azért nem váltotta még fel a CD-R-t, mert az üres CD-RW sokkal drágább az üres CD-R-nél. Ezenkívül a CD-R egyszeri írhatósága, letörölhetetlensége nagy előny a biztonsági mentéseknél.

2.3.10 DVD

Az CD/CD-ROM szabvány alapjait 1980-ban fektették le. A technológia fejlődött azóta, így nagyobb kapacitású optikai lemezek is gazdaságosan megvalósíthatók már és az igény is nagy rájuk. Hollywood nagyon szeretné az analóg videoszalagokat digitális lemezekre cserélni, hiszen ezek jobb minőségűek, olcsóbban előállíthatóak, tovább tartanak, kevesebb helyet foglalnak az üzletek polcain és nem kell őket visszatekerni. A szórakoztató-elektronikai társaságok is keresnek egy új szenzációs terméket, a szoftvergyártók pedig szeretnének termékeikhez multimédia-lehetőségeket adni.

A technológia és e három roppantul gazdag és rendkívül befolyásos iparág

igényének ilyen kombinációja vezetett a **DVD**-hez, amely eredetileg a **Digital Video Disk** (digitális video-lemez), most már a **Digital Versatile Disk** (digitális sokoldalúan felhasználható lemez) rövidítése. A DVD alapjaiban ugyanaz a konstrukció, mint a CD, 120 mm-es fröccsöntött polikarbonát lemez, amely lézer diódával megvilágított és fényérzékelővel olvasott mélyedéseket és síma felületeket tartalmaz.
Az újdonságok:

1. Apróbb mélyedések (0,4 mikron a CD-k 0,8 mikronjához képest)

2. Sűrűbb spirál (0,74 mikron a sávköz a CD-k 1,6 mikronjához képest)

3. Vörös lézer (0,65 mikronos hullámhossz a CD-k 0,78 mikronjához képest)

Ezek a fejlesztések együttesen a tárolókapacitást hétszeresére növelik, 4,7 GB-ra.
Az 1x DVD meghajtó 1,4 MB/s-on működik (szemben a CD 150 KB/s-ával). Sajnos a bevásárlóközpontokban is alkalmazott vörös lézerre történő átállás azt jelenti, hogy a DVD meghajtóknak szükségük van egy másik lézerre vagy különleges átalakító-optikára ,hogy a CD-ket is olvasni tudják, amelyre nem mindegyik képes. Ezenkívül a CD-R-ek és CD-RW-k DVD meghajtón történő olvasása nem biztos, hogy lehetséges.

Elég-e 4,7 GB? Talán. Az MPEG-2 tömörítő-eljárás (IS 13346-os szabvány) használatával egy 4,7 GB-os DVD-lemezre ráfér 133 perc teljes képernyős film 720x480-as felbontásban a hozzá tartozó hanganyaggal nyolc nyelven, további 32 nyelven feliratozással. Mindazonáltal néhány alkalmazás, mint pl. multimédiás játékok,kézikönyvek még ennél is több helyet foglalhatnak, a filmipar szeretne több filmet egy lemezre feltenni, ezért négy formátumot határoztak meg:

1. Egyoldalas, egyrétegű (4,7 GB)

2. Egyoldalas, kétrétegű (8,5 GB)

3. Kétoldalas, egyrétegű (9,4 GB)

4. Kétoldalas, kétrétegű (17 GB)

Miért ennyi formátum? Egy szóval: politika. A Philips és a Sony egyoldalas, egyrétegű lemezeket akart, a Toshiba és a Time Warner ezzel szemben kétoldalas, egyrétegű DVD-t szeretett volna. A Philips és a Sony szerint az emberek nem akarják a lemezeket fordítgatni egyik oldalról a másikra, a Toshiba és a Time Warner pedig nem hitt a kétrétegű lemez megvalósíthatóságában. Kompromisszum született: mind a 4 kombinációt szabványosították, a piac fogja eldönteni, melyik lesz sikeres.

A kétrétegű lemeznek van egy visszaverő réteg az alján, rajta egy félvisszaverő réteggel. Attól függően, hogy hova van a lézer fókuszálva, ugrál az egyikről a másikra.
Az alsó rétegen kissé nagyobb mélyedéseket és síma felületeket kell képezni, ezért tárolókapacitása valamivel kisebb, mint a felső rétegé.

A kétoldalas lemezek két 0,6 mm-es egyoldalas lemez összeragasztásával keletkeznek. Azért, hogy az összes változat ugyanolyan vastag legyen, az egyoldalas lemez is két 0,6 mm-es lemezből áll, amelyből az egyik alkalmatlan adatok tárolására (lehet, hogy a jövőben 133 percnyi hirdetést tesznek rá, hátha az emberek kiváncsiak

lesznek, mi van rajta). A kétoldalas, kétrétegű lemez szerkezetét a 2-27-es ábra mutatja.

	Polikarbonát anyag 1	
0,6 mm-es egyoldalas lemez		Félvisszaverő réteg
		Alumínium fényvisszaverő
	Ragasztóanyag	
		Alumínium fényvisszaverő
0,6 mm-es egyoldalas lemez		Félvisszaverő réteg
	Polikarbonát anyag 2	

2-27. ábra. Kétoldalas, kétrétegű DVD lemez.

A DVD-t egy 10 szórakoztató-elektronikai társaságot tömörítő konzorcium hozta létre. A társaságok közül hét japán, szoros együttműködésben a nagy hollywoodi stúdiókkal (amelyek közül néhányat a konzorcium japán tagjai birtokolnak).
A számítógépes és telekommunikációs ipar képviselőit nem hívták meg a "piknikre", ami azt eredményezte, hogy kibérelt mozikban mutatták be a DVD képességeit.
A DVD-lejátszó lehetőséget ad például a film káros jeleneteinek valós idejű kivágására (lehetőséget adva a szülőknek egy NC17-es osztályú film kisgyerekek számára is nézhetővé tételére) hatcsatornás hangminőségre ad lehetőséget, támogatja a Pan-and-Scan-t. A DVD-lejátszók újabban képesek eldönteni, hogyan vágják le a 3:2-es szélesség/magasság arányú filmek bal és jobb szélét, hogy azok kitöltsék a használatos TV-készülékek képernyőjét (amelyek szélesség/magasság aránya 4:3).

Egy másik dolog, amire a számítógépipar nem gondolt, a nemzetközi inkompatibilitás az Amerikának és az Európának szánt lemezek között, a többi kontinens szabványairól nem is beszélve. A filmipar azért igényelte ezt a funkciót, mert az új filmeket mindig az USA-ban mutatják be először és akkor szállítják Európába, amikor az USA-ban a videóváltozatot kiadják. Az ötlet az volt, hogy biztossá tegyék, hogy az európaiak ne vehessék meg az USA-ban kiadott videókat, ami csökkentené az európai mozik bevételét. Ha Hollywood kezében lenne a számítógépipar, talán 3,5 inches floppy-lemezek lennének az USA-ban és 9 cm-esek Európában.

Ha a DVD sikert arat, megindulhat a DVD-R (írható DVD) és a DVD-RW (újraírható DVD) tömeggyártása rövid időn belül. A DVD sikere azonban nem garantált, hiszen a kábeltársaságoknak teljesen más tervük van a filmek továbbítására. A csata már el is kezdődött.

2.4 INPUT/OUTPUT

Ahogyan a fejezet elején is említettük, egy számítógépes rendszernek három fő komponense van: a CPU, a elsődleges és másodlagos tárolók (memória, háttértárolók)

és az **I/O**(**Input/Output**) eszközök, mint pl. a nyomtatók, szkennerek és modemek. Eddig a CPU-t és a tárolókat vizsgáltuk. Most itt az ideje, hogy megvizsgáljuk az I/O eszközöket, és hogy ezek hogyan kapcsolódnak a rendszer többi részéhez.

2.4.1 Buszok (sínek)

Fizikailag a legtöbb PC-nek és munkaállomásnak a 2-28-as ábrához hasonló a szerkezete. A szokásos elrendezés egy fémdoboz(ház) egy nagy méretű nyomtatott áramkörrel az alján, amelyet alaplapnak hívnak. Az alaplapon van a CPU-lapka, néhány vájat (slot), melyekbe DIMM modulok illeszthetők és különböző segédlapkák. Ezenkívül tartalmaz egy buszt hosszában végigkarcolva és foglalatok (socket) amelyekbe kártyák illeszthetők. Előfordul, hogy két busz is van az alaplapon egy gyors (az újabb kártyáknak) és egy lassabb (a régebbi kártyák számára).

SCSI vezérlő
Hangkártya

Modem

Ház

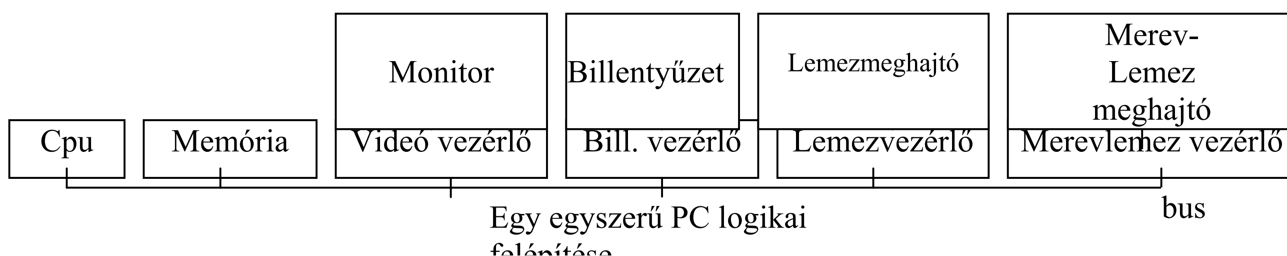
Csatlakozó

2-28. ábra. A PC fizikai szerkezete

Egy egyszerű PC logikai szerkezetét mutatja a 2-29. ábra. Ezen egyetlen busz köti össze a CPU-t, a memóriát és az I/O eszközöket; a legtöbb rendszer legalább két busszal rendelkezik. Mindegyik I/O eszköz két részből áll: az egyik, amelyik az elektronika nagy részét tartalmazza, a vezérlő(controller), a másik ami magát az eszközt tartalmazza, pl. egy lemezmeghajtót. A vezérlőt általában egy üres slotba dugott kártyán helyezik el, kivéve azokat, amelyek feltétlenül szükségesek (pl. billentyűzet-vezérlő), ezeket általában az alaplapon helyezik el. Bár a monitor is feltétlenül szükséges, mégis legtöbbször a videovezérlőt bedugható kártyán helyezik el, hogy a felhasználó választhasson a különböző videokártyák között (van-e grafikus gyorsító benne, mennyi a memóriája, stb.).A vezérlő a vezérelt eszközzel egy kábelen keresztül kapcsolódik, amelyet a ház hátulján lévő csatlakozóba dugnak.

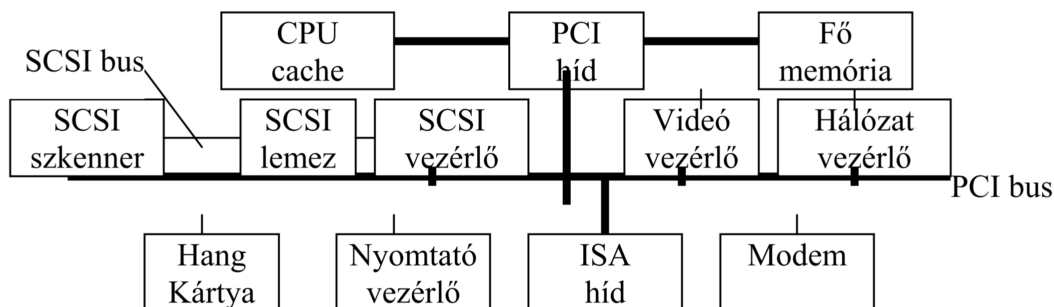
***[90-93]

SZÁMÍTÓGÉP FELÉPÍTÉSE



A vezérlő feladata, hogy irányítsa a hozzá kapcsolt I/O eszközök elérését. Ha egy program például adatokat akar egy lemezről, utasítást ad a lemezvezérlőnek, ami keresési parancsokat ad a meghajtónak. Ha a megfelelő track és sector megvan, a meghajtó egy bit sorozatot küld a vezérlőnek. A vezérlő feladata, hogy minden egység végén, ha az megtelt, megszakítsa a bit folyamot, és feltöltse az adatot a memóriába. Az egységek tipikusan egy vagy több szóból álnak. A vezérlő, amelyik adatokat ír a memóriába vagy onnan olvas a CPU beavatkozása nélkül közvetlen memória hozzáférésnek hívjuk, vagy a közismertebb néven DMA. Mikor az átvitel befejeződött, a vezérlő szabályos esetben megszakít, arra utasítja a CPU-t, hogy függessze fel az éppen futó programot és indítson el egy sajátos alkalmazást, a megszakítás kezelőt, hogy ellenőrizze a hibákat, és végrehajtson néhány szükséges alkalmazást, és értesítse az operációs rendszert, hogy az I/O művelet befejeződött. Mikor a megszakítás kezelő végzett, a CPU folytatja a megszakítás előtt félbehagyott programot. A bus-t nem csak az I/O vezérlő használja, a CPU is elhoz utasításokat, adatokat. Mi történik, ha a CPU és egy I/O vezérlő akarja használni a bus-t egyszerre? A válasz egy chip: bus arbiter (- bus "bíró"), amelyik eldönti, hogy ki következik előbb. Általában az I/O eszközök elsőbbséget élveznek a CPU fölött, mert a lemezeket és más mozgó eszközöket nem lehet leállítani, és várakoztatni, mert ez adatvesztéshez vezethet. Ha egy I/O sincs munkában, a CPU rendelkezik az összes bus ciklussal, amikre hivatkozhat a memóriában. Ha néhány I/O eszköz fut, a bus dönti el, mikor melyik eszköznek kap hozzáférést. Ezt az eljárást hívják "cikluslopásnak", ami lelassítja a gépet. Ez a koncepció jól működött az első PC-ken, mióta az összes alkatrész nagyjából egyensúlyban volt. Ahogy a CPU-k, memóriák, I/O eszközök felgyorsultak, a probléma felvetődött: a bus-nak ne legyen hosszú betöltési ideje. Egy zárt rendszerben, mint egy mérnöki munkaállomáson, a megoldás: tervezni kell egy új, gyorsabb bus-t az újabb modellekhez. Viszont senki sem szeretné eldobni I/O eszközeit - ha azok működnek – mikor a régi Memória bus :réli. Az emberek gyakran fejlesztik a processzorukat, de szeretnék tovább használni nyomtatójukat, szkennerüket, modemüket az új rendszeren is. Vegyünk egy nagy gyárat, kiépít egy széles körű szolgáltatásokat nyújtó I/O eszköz parkot IBM PC bus-hoz, nem lesz érdekelt abban, hogy az egészet eldobja, és újra kezdje az egészet. Az IBM rájött erre, és megalkotta az utódot, a PS/2-őt. A PS/2 egy új és gyorsabb bus-al rendelkezett, de sok gyár folytatta a régi bus-ok használatát, amit most már ISA (Ipari szabvány kiépítés) bus-nak hívtak. Sok lemez és I/O eszköz gyártó folytatta a régi bus használatát, ezért az IBM egy különös helyzetben találta magát, egyedül ő nem gyártott IBM kompatíbilis gépeket. Végül is ez arra kényszerítette, hogy visszatérjen az ISA bus-hoz. Mindemellett, a piac nyomására nem változott semmi, a régi bus tényleg lassú volt, szóval valaminek történnie kellett. Ebben a helyzetben más gyártók olyan gépeket fejlesztettek, amelyekben több bus volt, az egyik a régi ISA vagy az

EISA (bővített ISA). A legnépszerűbb közülük a PCI (periférikus alkotóelem összekapcsoló) bus. Ezt az Intel tervezte, és meghatározta, hogy az összes szabadalmat tegyék nyilvánossá, hogy a többi céget buzdítsa az átvételre.



A PCI bus-t sokféle konfigurációban lehet használni, egy tipikust lehet [illegible] rán.

Egy tipikus modern PC PCI és ISA bussal. A modem és a hangkártya ISA eszközök; SCSI vezérlő PCI eszköz

keresztül beszél. A vezérlő a memóriának és a PCI bus-nak közvetlenül üzen, tehát a túlzsúfolt CPU memória elkerüli a PCI bus-t. Ezért a gyors adatátviteli eszközök, mint a SCSI lemezek, közvetlenül a PCI bus-hoz tudnak csatlakozni. A PCI bus egy híd az ISA bus-hoz, hogy az ISA vezérlőket, és eszközeiket továbbra is lehessen használni. Egy ilyen gép három vagy négy üres PCI bővítőhelyet és ugyanennyi ISA bővítőhelyet tartalmaz, hogy a vásárlók használhassák egyaránt az ISA I/O kártyákat (általában a lassabb eszközökhöz) és az új PCI I/O kártyákat (a gyorsabb eszközökhöz).

Sok I/O eszköz áll ma az emberek rendelkezésére. A későbbiekben ezeket részletezzük.

Terminálok

A számítógép terminálok két fő részből állnak: a billentyűzet és a monitor. A nagyszámítógépek világában ezek a részek gyakran egy eszközben vannak integrálva, és kábelen vagy telefonvonalon csatlakoznak a fő géphez. A repülőtársaságoknál, a bankoknál és más nagyszámítógép orientált cégeknél ezek az eszközök széleskörűen elterjedtek. A PC-k világában a billentyűzet és a monitor független eszközök. Akár így, akár úgy a technológia ugyanaz.

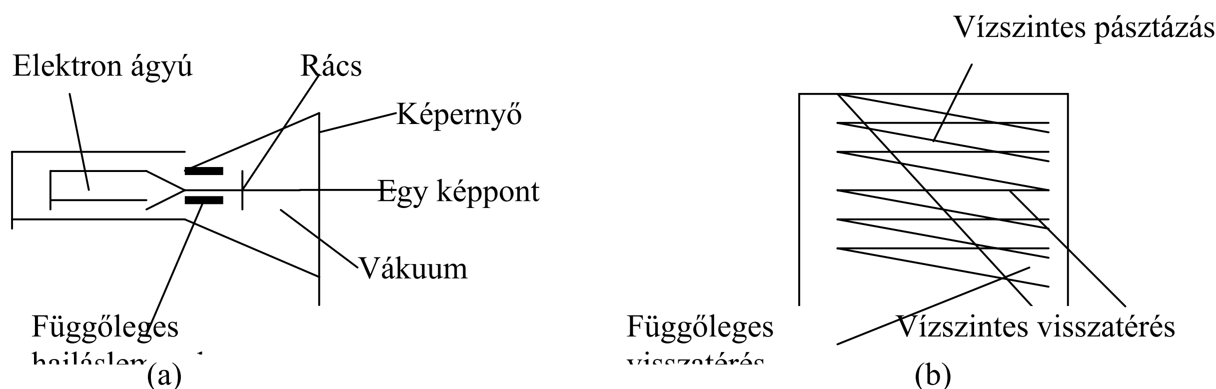
Billentyűzet

Többféle billentyűzet létezik. Az eredeti IBM Pc-nek olyan billentyűzete van, amelyik csattan egyet a billentyű lenyomása után. Napjainkban az olcsóbb billentyűzetek a gomb lenyomása után csak mechanikusan érintkeznek. A jobbakban a billentyű és az érintkező között van egy vékony gumiszerű anyag. Minden billentyű alatt van egy kis púp, ami behorpad a billentyű lenyomásakor. Egy kis pötty a púp belsején zárja az áramkört a billentyű lenyomásakor. Néhány billentyűzetben minden gomb alatt van egy kis mágnes, ami keresztül megy egy tekercsen a lenyomáskor, így egy pici feszültség keletkezik, ami észlelhető. Mind a mechanikus mind az elektronikus használatban van. Ha egy PC-n lenyomunk egy gombot, egy megszakítás keletkezik, és a billentyűzet megszakítás kezelő beindul (ez egy kis program, amit tartalmaz az operációs rendszer). A megszakítás kezelő kiolvassa a billentyűzetvezérlőben található hardvernyilvántartóból a lenyomott gomb számát (1-től 102-ig). Ha felengedjük a gombot egy második megszakító befejezi. Így ha a felhasználó lenyomja a SHIFT-et és utána az “m” betűt, aztán felengedi az “m”-et és a SHIFT-et is, a rendszer tudni fogja, hogy egy nagy “M”-et szeretnénk kiírni kis “m” helyett. Több gombos kezelést magába foglalja a szoftver a Shift-el, Ctrl-el és az Alt-

al kapcsolatban (például a CTRL-ALT-DEL billentyűkombináció, ami újra indítja az összes IBM PC és azokkal kompatíbilis gépeket.

CRT Monitorok

A monitor egy doboz, amiben egy CRT (Katódsugár cső) és ennek az áramellátása van. A CRT tartalmaz egy ágyút, ami elektron sugarat tud kilőni a cső végén lévő foszforeszkáló képernyőre (a színes monitorokban három elektron ágyú van, egy a piros, egy a zöld és egy a kék színnek). Vízszintesen a sugár kb. 50 μsec. alatt megy végig, és így kijelöl minden vízszintes vonalat a képernyőn. Ha végrehajtotta az egész képernyőn, visszamegy a bal felső sarokba, hogy újra kezdje a pásztázást. Azt az eszközt, amelyik sorról sorra rajzolja ki a képet raszterpásztázó eszköznek hívják (pl. TV).



(a) A CRT keresztmetszete. (b) A CRT pásztázó

A vízszintes pásztázást egy lineárisan növekvő feszültség vezérli, hogy a vízszintes hajláslemezek az elektron ágyú bal és jobb oldalára kerüljenek. A függőleges mozgást egy sokkal lassabban növő feszültség irányítja, hogy a függőleges hajláslemezek az ágyú alá és fölé kerüljenek. 400-1000 pásztázás után a feszültség a vízszintes és a függőleges hajláslemezeket gyorsan egymásba fordítja, hogy visszakerüljön a sugár a bal felső sarokba. Egy teljes képernyős képet szabályos esetben 30-60-szor frissít percenként. Bár mi úgy jellemeztük a CRT monitorokat, hogy elektronikus mezőt használnak a képernyő végigpásztázásához, de sok monitor mágneses mezőt használ az elektromos helyett, legfőképpen a high-end monitorok. Ahhoz, hogy a képernyőn pontokból egy minta keletkezzen, a rács rajta van a CRT-n. Amikor pozitív feszültség éri a rácsot, az elektronok felgyorsulnak, és a sugár nekimegy a képernyőnek, hogy röviden felizzón. Ha a töltés negatív, a sugár nem éri el a rácsot, és az nem izzik fel. Így az alkalmazott feszültségtől függ, hogy milyen minta jelenik meg a képernyőn. Ez a mechanizmus teszi lehetővé, hogy a bináris villamos jeleket átalakítsák sötét és világos pontokból álló képi jelekké.

***[94-97]

Kiss István, 94.o.-97.o.

Lapos monitorok

A CRT monitorok túlsagosan terjedelmesek és nehezek a hordozható számítógépek használatához, így teljesen eltérő technológia szükséges a hordozható számítógépek képernyőihez. Az egyik legáltalánosabban használt technológia az **LCD (Liquid Crystal Display : Kristályfolyadékos kijelző).** Ez nagyon összetett, sok fajtája van és gyorsan változik, így ez a leírás szükségszerűen rövid és leegyszerüsített lesz.

Nyúlós, szerves molekulákból áll a kristályfolyadék, ami folyik, mint egy folyadék, de térbeli szerkezete is van, mint a kristálynak. Egy osztrák botanikus (Rheinitzer) fedezte ezt fel 1888-ban és először az 1960-as években alkalmazták kijelzőknél (pl.: zsebszámológépeknél, óráknál). Amikor az összes molekula egy vonalban azonos szögben áll, akkor a kristály optikai tulajdonságai a bejövő fény szögétől és sarkításától függnek. Egy elektromos mezőt alkalmazva megváltoztatható a molekulák csoportosulása és ezen keresztül az optikai tulajdonságaik. A gyakorlatban ez azt jelenti, hogy a kristályfolyadékon keresztülragyogó fény kimeneti erőssége elektronikusan szabályozható. Ezt a tulajdonságot használják ki a lapos monitorok felépítései.

Egy LCD monitor képernyője két párhuzamos üveglapból áll, amik maguk között hangszigetelten tartalmazzák a kristályfolyadékot. Átlátszó elektródák vannak mindkét üveglaphoz hozzákapcsolva. A fény a hátsó üveglap mögül (természetesen vagy mesterségesen) hátulról világítja meg a képernyőt. Az átlátszó elektródák mindkét üveglaphoz hozzákapcsolva elektromos mezőt hoznak létre a kristályfolyadékban. A képernyő különböző részei eltérő feszültséget kapnak, hogy így szabályozzák a kép megjelenését. A képernyő elejéhez és hátuljához polaroidok vannak ragasztva, mert a megjelenítés technikájához sarkított fény szükséges használni. Az általános felépítést mutatja a 2-32(a) ábra.

Habár számos fajtáját használják az LCD monitoroknak, mi most egy különleges monitorfajtát, a **Tn (Twisted Nematic)** monitort vesszük példának. Ennél a monitornál a hátsó üveglapon vízszintes, míg az elülső üveglapon függőleges apró barázdák találhatók, mint ahogy a 2-32(b) ábrán is látható. Az elektromos mező hiányában az LCD molekulák a rovátkák iránya szerint rendeződnek sorba. Mivel így az elülső és a hátsó rendeződés eltér 90 fokkal, ezért a molekulák (és ennek következtében a kristály szerkezete) megcsavarodik előről hátra felé.

A monitor hátsó felén vízszintes polaroid van. Ez csak vízszintesen sarkított fény enged át. A monitor elején függőleges polaroid van. Ez csak függőlegesen sarkított fényt enged át. Ha nem lenne folyadék az üveglapok között, akkor a hátsó polaroid által beeresztett vízszintesen sarkított fényt elzárná az elülső polaroid és így a képernyő egységesen fekete lenne.

Kristályfolyadék
Hátsó üveglap Elülső üveglap
Hátsó elektróda Elülső elektróda
Hátsó polaroid Elülső polaroid

sötétség

Fényforrás fény

(a) (b)

2-32. ábra (a) Egy LCD képernyő szerkezete. **(b)** A hátsó és az elülső üveglapon lévő, egymásra merőleges barázdák.

Azonban az LCD molekulák csavart szerkezete vezeti a fényt és ahogyan az halad és forog rajtuk, vízszintesen jön ki. Ennek következtében és egy elktromos mező hiányában az LCD képernyő egységesen világít. Az üveglap kiválasztott részein lévő feszültség által a csavart szerkezet megsemmisülhet, így elzáródik a fény azokon a részeken.

Két módszert használnak általában a feszültség létrehozására. A **passzív mátrix display** -nél (ez az olcsóbb módszer) mindkét elektróda tartalmaz párhuzamos drótokat. Például a 640X480-as felbontásnál a hátsó elektróda 640 függőleges, míg az elülső 480 vízszintes drótot tartalmaz. Azáltal, hogy áram kerül a függőleges drótok egyikére és ezalatt egy lüktetésre az egyik vízszintesre is, a kiválasztott pixel (képpont) helyén megváltozik a feszültség, így elsötétül az egy pillanatra. Ismételgetve ezt a lüktetést a következő, majd az azt követő pixelle egy sötét scan line (átfutó sor) tehető ki, hasonlóan, ahogy egy CRT dolgozik. Rendes körülmények között a teljes képernyő másodpercenként 60-szor tehető ki, így becsapva a szem azt látja, hogy egy mozdulatlan kép van a képernyőn, szintén úgy, ahogy a CRT-knél.

A másik, szintén széles körben használt módszer az **aktív mátrix display**. Ez meglehetősen költségessebb, de jobb képet nyújt, így ez a nyerő terület. Ennél a módszernél ahelyett, hogy csak két irányban lennének merőleges drótok, az egyik elektródán minden pixel helyén apró kapcsolók vannak. Ezek be- és kikapcsolásával tetszőleges feszültségminta jöhet létre az egész képernyőn, figyelembe véve egy tetszőleges bit mintát.

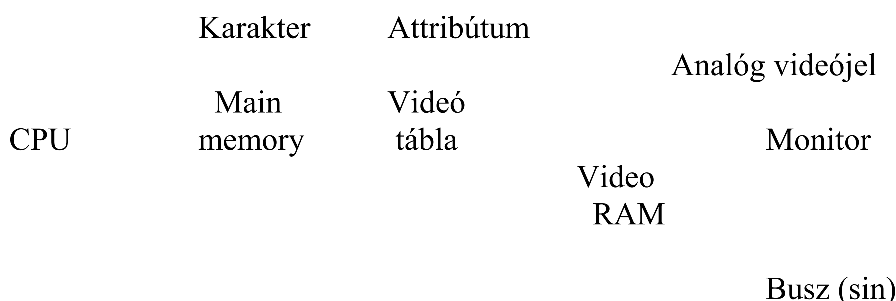
Így végül van egy leírásunk, hogy hogyan működnek a fekete-fehér monitorok. A színes monitoroknál elég annyi, hogy általában hasonló alapelven működnek, de azok részleteikben jóval bonyolultabbak. Minden pixel helyén optikai szűrőket használnak szétválasztani a fehér fényt a vörös, a zöld és a kék összetevőjére, hogy azokat egymástól függetlenül lehessen megjeleníteni. Minden szín felépíthető ennek a három alapszínnek az egymásra vetítésével.

Alfanumerikus terminálok (character-map terminálok)

Általában a terminálok három fajtáját használjuk : alfanumerikus terminál, grafikus terminál és RS-232-C terminál. Ezek mindegyikét akármilyen billentyűzettípusról lehet használni, de a számítógép velük való kommunikációs módjában és az output kezelésében különböznek. Mi most leírjuk mindegyik fajtát.

Egy személyi számítógépnél két lehetőség van, hogy meghatározzuk a képernyőre kerülő outputot : az alfanumerikus és a grafikus terminál segítségével. A 2-33. ábra azt mutatja, hogy az alfanumerikus terminál hogyan jeleníti meg az outputot a monitoron. (A billentyűzet teljesen eltérő módon van kezelve.) A video tábla két

részből áll : egy nagy memóriából, amit **videó memoriának (video memory)** hívunk, és egy kevés elektronikából a bemenő busznak és a videójelek generálására.



2-33.ábra A terminál outputja a személyi számítógépen.

A CPU átmásolja a karaktereket a megjelenítésükhöz a videó memoriába. Minden karakterhez tartozik egy **attribútum byte (jelző byte),** hogy az tartalmazza, hogy a karakter hogyan jelenjen meg. Az attribútomok tartalmazhatják a karakterek színét, fényerősségét vagy akár, hogy villogjon a karakter és így tovább. Ezért egy 25X80 karakterből álló kép 4000 byte-on tárolható a videómemóriában, 2000 byte-on a karakterek és 2000 byte-on az attribútomok. A legtöbb videótáblának ennél több memóriája van, hogy több képernyőképet is tárolhasson.

A videótábla feladata, hogy folyamatosan kiolvassa a karaktereket a videó RAM-ból és generálja a szükséges jeleket a monitorvezérlőnek. Egyszerre egy teljes sornyi karaktert olvas be, így kiszámítható az egyedi jelek sora. Ez a jel egy magas frekvenciájú analóg jel, ami szabályozza az átfutó fényelektront, hogy hogían jelenítse meg a karaktereket a képernyőn. Mivel a tábla kimenetelei videójelek, ezért a monitorban muszály lennie néhány mérőórának a számítógépből, hogy azok megakadályozzák az eltorzulást.

Grafikus terminálok (bit-map terminálok)

A grafikus terminél ötlete azért merült fel, hogy a képernyő ne olyan legyen, mint egy 25X80-as karakterekből álló tömb, hanem olyan legyen, mint egy kép alkotóelemeiből álló tömb, ezeket a képpontokat hívjuk **pixelnek.** Minden pixel be- vagy kikapcsolt állapotban van. Ez az információ egy biten tárolható. A személyi számítógépek képernyőjén lehet mindössze 640X480 pixel, de általánosabb a 800X600 vagy még több pixel. A mérnöki munkaállomások képernyői tipikusan 1280X960 pixelből vagy még többől állnak. A terminálokat gyakrabban használják grafikusan, mint alfanumerikusan, és ezt grafikus terminálnak (bit-map terminálnak) hívjuk. Minden modern videótábla teljesen úgy működik, mint az alfanumerikus vagy a grafikus terminál, egy szoftver irányítása alatt.

Itt ugyanazt az általános ötletet használják, mint ami a 2-33.ábrán van, kivéve, hogy

a video RAM csak egy nagy bithalmaznak látszik. A kezelő szoftver bármilyen mintázatot be tud állítani, ami csak kellhet és az azonnal meg is jelenik a képernyőn. A karakterek megjelenítésénél a szoftver határozhatja meg, hogy pl. egy 9X14-es táglalapon ábrázol minden karaktert és kitölti a karakterhez szükséges biteket, hogy az megjelenjen. Ez a módszer megengedi a szoftvernek, hogy különböző betütípusokat készítsen és azokat betükészletté állítsa össze. Minden hardver meg tudja jeleníteni a bittömböt. Színes megjelenítés esetén minden pixel 8, 16 vagy 24 bitből áll.

Általában arra használják a grafikus terminálokat, hogy azok segítsék tárolni a különálló **ablakok (windows)** megjelenítését. Az ablak a képernyőnek azon területe, amit egy program használ. Több ablak lehetővé teszi, hogy különböző programok fussanak azonos időben és minden egyes megjelenítés eredménye független a többitől.

Habár a grafikus terminálok nagyon rugalmassak, mégis van két nagy hátrányuk. Először is tekintetes részre van szükségük a video RAM-ból. Napjainkban a legáltalánosabb méretek a 640X480 (VGA), 800X600 (SVGA), 1024X768 (XVGA) és az 1280X960. Észrevehetjük, hogy mindegyiknél megegyezik az oldalak aránya (szélesség : magasság) 4:3, hogy ez hasonló legyen az általánossan elterjedt tévé oldalainak arányaihoz. Ahhoz, hogy igazi színeket kapjunk, 8 bit szükséges mind a 3 alapvető színhez, azaz 3 byte kell pixelenként.Ennek következtében egy 1024X768-as monitorhoz 2,3 MB video RAM szükséges.

A nagy igény miatt néhány számítógép megelégszik egy 8 bites szám használatával a színkövetelményhez. Ez a szám egy indexként van tárolva a hardvertáblában, amit **színpalettának (color palette)** hívnak, és ez 256 ilyen bejegyzést tartalmaz, amelyek mindegyike egy 24 bites RGB értéket tartalmaz. Ez a módszer, amit indexelt színnek (indexed color) hívunk, 2/3-ára csökkenti a video RAM beli igényt, de csak 256 színt engedélyez a képernyőn egyszerre. Általában a képernyőn minden ablaknak van saját színtérképe, de csak egy színpaletta van. Gyakran, amikor több ablak van egyszerre a képernyőn, csak az aktuálisban van lehetőség beállítani a színeket.

A második hátránya a grafikus megjelenítésnek a megvalósítása. A programozók minden pixelt úgy változtathatnak egyszerre mindkét helyen és időben, ahogy akarják. Igaz, hogy a monitorra kimásolhatóak az adatok a video RAM-ból annélkül, hogy keresztülhaladnának a fő rendszerbuszon, de az adatok bekerüléséhez a video RAM-ba a fő rendszerbusz szükséges. Egy egész képernyős, teljesen színes multimédiás film megjelenítéséhez a képernyőn, egy 1024X768-as felbontásban a videó RAM-ba 2,3 MB adatot kell bemásolni filmkockánként. Egy mozgó videófilmnek legalább 25 filmkocka kell másodpercenként, így a teljes adatmozgás 57,6 MB/sec. Ez a teher jóval nagyobb, mint amennyit az (E)ISA busz tud kezelni, így az IBM PC-ken magas teljesítményű videókártyákra, PCI kártyákra van szükség, és még ekkor is nagy kiegyezés kell.

A teljesítményprobléma összekapcsolható a képernyő gördülésével. Az egyik lehetőség, hogy minden bitet másolunk a szoftverben, de ez oriási terhet tesz a CPU-ra.

***[98-101]

Nem meglepő módon sok videokártyát azért építettek speciális hardverrel, hogy másolás helyett a bázisregiszterek megváltoztatásával lehessen mozgatni a képernyő részeit.

2.2.4 RS-232-C terminálok

Sok cég számítógépeket készít, és sokan terminálokat (különösen szerverekhez). Ahhoz, hogy (majdnem) minden terminált (majdnem) minden számítógéphez lehessen használni, az EIA (Elektronikai Ipar Szövetség) egy egységes számítógép terminál interfészt (amit RS 232-C-nek hívnak) fejlesztett ki. Bármely terminál amelyik támogatja az RS-232-C interfészt, össze lehet kapcsolni bármely számítógéppel, ami ugyancsak támogatja ezt az interfészt.
Az RS-232-C terminálóknak 25 tűs csatlakozójuk van. Az RS-232-C szabvány előírja a csatlakozók mechanikai méreteit és alakját, a feszültség szintet és a csatornákon lévő jelek jelentését.
Ha a számítógép és a terminál távol vannak egymástól, akkor jut fontos szerephez, mert ez az egyetlen praktikus megoldás összekapcsolni őket, a telefonhálózaton kívül. Sajnos a telefonhálózat nem alkalmas azon jelek átvitelére, amelyeket az RS-232-C szabvány előír, modemnek (modulátor-demodulátor) nevezik azt az eszközt, amit a számítógép és a telefon, vagy a terminál és a telefon közé helyeznek, hogy az elvégezze a jelek átalakítását. Mi a modemről később fogunk tanulni.
A 2-34 ábrán látható a számítógép, a modemek és a terminál elhelyezkedése, ha telefonhálózatot használnak. Ha a terminál annyira közel van a számítógéphez, hogy össze lehet kábellel kötni, akkor modemeket nem használnak, hanem inkább RS-232-C csatlakozókat és kábeleket, ilyenkor nincsen szükség a modem ellenőrző tűire.
Ahhoz, hogy kommunikálni tudjon, a számítógép és a terminál is tartalmaz egy csipet, amit UART-nak neveznek (Univerzális Asszinkron Vevő és Továbbító), ez logikailag eléri a buszt. Ahhoz, hogy megjelenítsen egy karaktert, a számítógép beolvassa a karaktert a főmemóriából és az UART-nak küldi át, ami ezután továbbítja az RS-232-C kábelnek bitenként. Gyakorlatban az UART egy párhuzamos-soros átalakító, mivel egy egész karaktert (1 bájt) kap egyszerre és egyszerre egy bitet ad ki rendszeres ütemben. Ezen kívül hozzácsatol Egy start és egy stop bitet egy karakterhez azért, hogy a karakter kezdetét és végét behatárolja (110 bps-enként két stop bitet hozzácsatol).
A terminálokon egy másik UART kapja a biteket, és felépíti az egész karaktert, amelyet aztán kivetít a képernyőre. A terminál billentyűzetéről bemenő adat párhuzamos-soros átalakításon megy keresztül a terminálban és aztán az UART újra összegyűjti ezeket a számítógépnek.
Az RS-232-C szabvány meghatároz több mint 25 jelet, de a gyakorlatban csak egy párat használnak, és azoknak a legtöbb részét el lehet hagyni, amikor a terminált közvetlenül a számítógéphez kötik modemek nélkül. A második és harmadik tű azért van, hogy továbbítsa és fogadja az adatokat. Mindegyik tű egy irányú bitsorozatot kezel, amikor a terminál és a számítógép be van kapcsolva akkor a számítógép megerősítést ad (Data terminal ready). Amikor a terminál és számítógép adatokat akar küldeni, megerősíti az adatfogadásra kérés (Request to Send) jellel. Ha a modem megadja az engedélyt, megerősíti ezt a küldésre készen áll (Clear to Send) jellel. Más tűket különböző státusz, tesztelő és időzítő funkciókra használnak.

2.3.4 Egér

Ahogy az idő telik az emberek egyre kevesebb szakértelemmel is használják a számítógépeket. Az ENIAC generáció számítógépeit csak azok használták, akik építették azokat. Az ötvenes években a számítógépeket csak magas képzettségű hivatásos programozók használták. Mostanában a számítógépeket széles körben használják azok az emberek is, akiknek el kell végezni valami munkát, és akik nem tudnak vagy nem is akarnak sokat tudni arról, hogy hogyan működik a számítógép és hogyan van programozva.

Régen a legtöbb számítógépnek volt parancsvonal menürendszere, amelybe a felhasználók begépelték a parancsokat. Mivel azok az emberek, akik nem voltak számítógép specialisták gyakran a parancsvonal menürendszert úgy érezték, hogy nem volt felhasználóbarát, vagy még ennél is sokkal rosszabbra gondoltak, sok számítógép gyártó kifejlesztett egérrel kezelhető menürendszert (pl.: Macintosh és Windows). Ahhoz ezt a módszert használhassuk szükség, van olyan rendszerre ami ezt megjeleníti a képernyőn. Az egérrel történik a felhasználók számára a legegyszerűbb módon. Az egér egy kis műanyag doboz és a billentyűzet mellett helyezkedik el. Amikor az egérpadon mozgatják egy kis mutató a képernyőn mozog és lehetővé teszi a felhasználónak, hogy képernyőn megjelenített szövegekre klikeljen. Az egérnek egy, kettő vagy három gombja van és ezzekel a felhasználó a menüből tud választani. Sok vita folyt azon, hogy hány gomb legyen az egéren. Naiv felhasználók az egy gombost részesítik előnyben (mivel nehéz rossz gombot nyomni rajta), de a tapasztaltabbak szeretik a több gombos megoldást, hogy több műveletet lehessen az egérrel elvégezni.

Három fajta egeret készítettek: mechanikus, optikai és opto-mechanikus egeret.

Az első egereknek két görgője van amelyikek kilógtak az egér alján, és a tengelyei derékszöget zártak be. Amikor az egeret a főtengelyével párhuzamosan mozgatták akkor az első görgő mozgott, amikor a főtengelyre merőlegesen akkor a másik mozgott. Mindkét görgő mozgatott egy potmétert. Az ellenállás változás megmérésével lehetséges volt azt kiszámolni mennyit forogtak el a görgők, és így tudták mennyit mozdítottak el az egéren. Az utóbbi években ezt az egérfajtát átalakították olyanra, hogy egy golyó van a görgők helyett (Lásd 2-35).

Az egerek második fajtája az optikai egér ilyen van nekem is. Ennek se golyója se görgője nincs. Ehelyett LED (fényt kibocsátó dióda) van benne és egy fényérzékelő. Az optikai egeret egy speciális műanyag padon használják, amely négyzetrácsot tartalmaz, amelyen sűrűn vannak vonalak. Ahogy az egér mozog a négyzetrács felett a fotóérzékelő érzékeli a vonalakat úgy hogy érzékeli a fénynyalábokat amik a padról visszaverődnek. Az egérben lévő elektronika kiszámítja az elmozdulásnak nagyságát a padon.

A harmadik az optomechanikus. Hasonlóan az újabb mechanikus egerekhez ebben is egy golyó van amely két tengelyt forgat, amelyek merőlegesek egymásra. A tengelyekre két tárcsa van erősítve amiken olyan lyukak vannak, amiken a fény átjuthat. Amikor az egér mozog, akkor a tengelyek forognak, és a fény átjut az érzékelőbe, ha a tárcsán egy lyuk kerül a LED és az érzékelő közé. Az érzékelt impulzusok száma egyenesen arányos a mozgás mennyiségével.

Habár az egeret sokféleképpen lehet üzemeltetni, egy közkedvelt módszer az, hogy az egérrel három bájtos adatsorozatokat küldetnek a számítógéphez minden alkalommal, amikor az egér egy bizonyos minimális távolságot mozdul (pl.: 0,01 ’’), amit néha

mickey-nek neveznek. Rendszerint ezek a bájtok soros vonalon bitenként érkeznek. Az első bájt tartalmaz egy egész számot, mely megmutatja mennyit mozdult az egér az x tengelyen 100 ms alatt. A második bájt ugyanezt az információt adja az y koordinátáról. A harmadik bájt tartalmazza az egér gombjainak aktuális állapotát. Néha két bájtot használnak minden koordináta meghatározására.
A számítógép alacsony szintű szoftvere fogadja ezeket az adatokat, és átalakítja a relatív mozdulatokat, amelyeket az egér küld egy abszolút pozícióból. Ezután egy nyilat kijelez a képernyőn abban a pozícióban, amely megegyezik az egér helyzetével. Amikor a nyíl a megfelelő menüpontra mutat, a felhasználó klikkel egyet az egérrel és a számítógép ez után meg tudja állapítani, hogy melyik menüpont lett választva abból.

2.4.4 Nyomtatók

Ha készítünk egy dokumentumot, vagy letöltünk a www-ről egy oldalt, azt gyakran ki akarjuk nyomtatni, így minden számítógépet fel lehet szerelni nyomtatóval. Ebben a részben le fogjuk írni a monokróm és színes nyomtatók néhány fajtáját.
Monokróm nyomtatók
A leggyakoribb a mátrixnyomtató , melyben egy nyomtatófej, amely 7 és 24 között elektromágnesen aktivizálható tűt tartalmaz, jár minden vízszintes nyomtatási vonalon. A legegyszerűbb nyomtatóknak hét tűjük van, és 80 5x7 méretű karaktert tud egy sorba kinyomtatni. A nyomtatási vonalban 7 sor és 5x80=400 pont van. Minden pontról el lehet dönteni, hogy kinyomtassa vagy ne. A 2-36(a) ábra egy “A” betű nyomtatott képét illusztrálja 5x7 mátrixban.
Két féleképpen lehet a nyomtatási minőséget javítani: több tű felhasználásával vagy kör átfedéssel. A 2-36(b) ábra mutatja a “A” képét 24 tűs nyomtatóval, amely átfedési pontokat is tartalmaz. A megnövelt minőség lassabb nyomtatási sebességgel jár. A legtöbb mátrixnyomtató többféle módon működhet, különböző módokat kínálva, hogy a minőség vagy a sebesség a fontos. A mátrixnyomtatók olcsóak (főleg az átlagfelhasználónak), megbízhatóak, de lassúak, hangosak és kicsi a felbontásuk. Három fő felhasználói módszer van…

Ábrák:
2-34.
Szignálok:
-GND
-Átvitel
-Vétel
-Átvitelkérés
-Adatküldésre kész
-Adatelérés rendben
-Közös visszatérés
-Vivő felderítő
-Adat terminál készen áll

***[102-105]

Egyrészt gyakran használják nagy előnyomtatott űrlapok nyomtatásához. Másrészt jól lehet vele nymtatni kisebb papirokra, úgymin a pénztárgépszalagok, bankjegy automata, vagz hitelkartya tranzakciós cédulája, vagy repülőgépen az étkezési jegyek. Harmadrészt, nyomtatható többrészes folytonos űrlap indigóval együtt. Általában ez a legolcsóbb technológia.

Otthoni alacsony költségű nzomtatásban a **tintasugaras nymtató** a kedvelt. A mozgatható nyomtatófej, amely egy tintapatront tart vízszintesen mozog a papír fölött, miközben a tintát kifújja a kis fúvókákón. Minden fúvókában egy kis csepp tintát elektromos áramall melegítenek a forráspontig, amíg a tintacsepp ki nem tör. A tintacsepp így a fúvókán át a papírra kerül. Ezután a fúvókát lehűtik és ay így keletkező vákum egy újab tintacseppet szippant be. A nyomtató sebességét az adja meg, hogy milyen gyorsen lehet egy melegítés - hűtés ciklust megismételni. A tintasugaras nyomtatók felbontása általában 300 és 720 dpi (dot per inch) között van, de vannak 1440 dpi-s nyomtatók is. Előnye, hogy olcsó, csendes és jó minőségben nyomtat, viszont lassú, drága tintapatront használ és tintával eláztatott papírt adnak ki.

Talan a legizgalmasabb fejlesztes a nyomtatasban azota, hogy Guttenberg Janos feltalalta a konyvnyomtatast a XIX. szazadban, a **lezernyomtato**. Ez az eszkoz kombinalja a jo minosegu kepet, a rugalmassagot, jo sebesseget es egy elfogadhato arat. Mindez egy onallo periferia. A lezernyomtatok szinte ugyanolyan technikaval dolgoznak, mint a fenymasologepek. Valojaban sok ceg keszit olyan eszkozt ami masol is es nyomtat is (es neha meg faxol is).

A mukodest nagyvonalakban a 2-37. abra mutatja. A nyomtato szive egy forgo precizios dob. Ez minden oldalkezdesi ciklusnal feltoltodik 1000 volttal es fenyerzekeny anyaggal vonodik be. Ezutan egy lezernyalab pasztazza vegig a dobot hosszaban (hasonloan az elektronsugarhoz a TV-keszulekekben). A vizszintes elmozdulast egy forgathato nyolcoldalu tukorrel oldottak meg. A fenysugar vilagos es sotet pontokbol allo mintat alkot. Az a pont amit a fenysogar er elveszti az elektromos tolteset.

Egy sor pont festese utan a dob fordul egy kicsit, igy a kovetkezo sor lesz festheto. Egyszer csak az elso sor eleri a tonert, egy elektromossagra erzekeny fekete port tartalmazo tartalyt. Azok a pontok amelyek meg nem vesztettek el toltesuket, magukhoz vonzzak a tonert, igy egy lathato sor keletkezik. Kicsivel kesobb a toner boritotta dob ranyomodik a papirra, atnyomva a fekete port. Ezutan a papir futott gorgokon halad keresztul, hogy a toner folyamatosan raolvadjon a papirra. Ha a dob teljesen korbefordult, elvesziti elektromos tolteset, megtisztul a maradek tonertol, es keszen all a kovetkezo oldal nyomtatasara.

Ezek a muveletek bonyolult kombinacioi a fizikanak, kemianak, mechanikai tervezesnek, optikai tervezesnek, es igy nehez is beszelni rola. Azonban szamos elado kinal elkeszult szerkezeteket, un. nyomtato-motorokat. A lezernyomtatokat gyarto cegek a nyomtatomotorokat a sajat elktronikajukkal, es szoftverrukkel kombinaljak, es igy kesitik el sajat nyomtatojukat. Az elktronika tartalmaz egy processzort, es tobb megabyte memoriat, hogy tarolni tudjon egy teljes oldalt bitterkepesen es a beepitett es a letoltheto fontkeszleteket. A legtobb nyomtato elfogadja a nyomtatasi parancsot, ami csak annyit mond, hogy nyomtasson egy oldalt (ellentetben azokkal a nyomtatokkal amik a fo CPU altal elkeszitett bitterkepet nyomtatjak). Ezek a parancsok kulonbozo nyelveken adhatok ki, mint peldaul a PCL (HP) vagy a PostScript (Adobe).

A lezernyomtatok felbontasa 600 dpi vagy nagyobb, es jo minosegu fekete-feher kepet lehet vele nyomtatni, de ez bonyolultabb mint elso ranezesre tunik. Vegyunk peldaul egy fenykepet amit 600 dpi-vel scanneltunk be es 600 dpi-vel nyomtatunk ki. A beszkennelt kep 600x600 pixelt tartalmaz inchenkent es egy szurkesegi arnyalatot 0-tol (feher) 255-ig (fekete). A nyomtatonk is 600 dpi-s, de minden egyes pixel vagy feher, vagy fekete. Szurket nem tud nyomtatni.

A szokasos megoldas a szurke arnyalatok hasznalatara a **halftoning** (felarnyalt) eljaras, olyan mint a reklamposzterek. A kepet 6x6 pixelbol allo cellakra bontjak. Minden cella 0 es 36 kozott tartalmaz fekete pixeleket. A szem sotetebbnek erzekel egy cellat, ha tobb benne a fekete pont. A szurke ertekek 0 es 255 kozott lehetnek melyek 37 zonat alkotnak. 0-tol 6-ig a 0. zona, 7-tol 13-ig az 1. zona es igy tovabb (lathato, hogy a 36. zona kisebb mint a tobbi, mert 256 nem osztheto 37-tel maradek nelkul). Ha a szurke erteke megegyezik a 0. zonaval, akkor a papiron a cella helye ures marad (2.-38.a,). Az 1. zona erteke 1 fekete pixel. A 2. zona erteke 2 fekete pixel, es igy tovabb (2.-38.c,-f,). Termeszetesen egy 600 dpi-vel szkennelt kep felarnyaltjanak effektiv felbontasa csokken 100 cella/inch-re (halftone screen frequency), hagyomaros mertekegysege az lpi (lines per inch).

Szines nyomtatok

Szines kep ketfelekeppen allithato elo: additiv fennyel, es szubtraktiv fennyel. Az additiv szinkeveressel keszult kepek (olyan mint a CRT monitorok kepe) a harom elsodleges additiv szin (voros, zold, kek) linearis szuperpoziciojabol epulnek fel. A szubtraktiv szinkeveressel keszult kepek (mint a szines fotok egy magazinban) bizonyos hullamhosszu fenyt elnyelnek a maradekot visszaverik. Ezek a harom elsodleges szubtraktiv szin, a cyan (turkiz) (minden voroset elnyel), a sarga (minden keket elnyel) es a magenta (lila) (minden zoldet elnyel) linearis szuperpoziciojabol epul fel. Elmeletileg minden szin eloallithato a turkiz, sarga, es lila szinekbol. Gyakorlatban viszont nehez ezeket a szineket egyenletesen elkeverni ugy, hogy minden fenyt elnyeljen, es tiszta fekete szint adjon. Emiatt majdnem minden nyomtato negyfele tintat hasznal: turkiz, sarga, lila, es fekete. Az ilyen rendszeru nyomtatokat CYMK nyomtatoknak hivjuk. (Cyan Yellow Magenta blacK.) Ezzel ellentetben a monitorok az additiv szinkeverest hasznaljak, es RGB rendszeruek. (Red Green Blue.)

Azt a szinmennyiseget amit egy display vagy printer kepes eloallitani **szinmelysegnek** nevezzuk. Nincs olyan eszkoz aminek a szinmelysege megegyezne a valos vilageval. Eddig a legjobb, hogy minden szinnek 256 intenzitasa van, de ez meg csak 16,777,216 szin. A technologia hianyossaga, hogy lecsokken az osses szin szama, es a fennmardo szineket nem mindig lehet egyenletesen kitolteni. Raadasul nem csak fenytannal, hanem a retina mukodesevel is foglalkozni kell.

A fenti obszervaciok konzekvenciaja, hogy kepernyon jol kinezo kep konvertalasa egy vele megegyezo nyomtatott formava, kozel sem egyszeru. A problemak tobbek kozott:

1. A szines monitorok additiv szinkeverest hasznalnak, mig a szines nyomtatok szubtraktiv szinkeverest
2. A CRT monitorok 256 intenzitast hasznalnak szinenkent, a szines nyomtatok felarnyaltak
3. A monitoroknak sotet hattere van, mig a papirnak vilagos
4. Az RGB es a CYMK szinmelysegek kulonbozoek

Egy valosagos keppel megegyezo kep nyomtatasahoz szukseges: eszkoz kalibralas, kifinomult szoftver es jelentos szaktudas a felhasznalo reszerol.

Otfele technologiat hasznalnak a szines nyomtatashoz a mindennapokban. Mindegyik a CMYK rendszeren alapszik. Ilyenek a szines tintasugaras nyomtatok. Ezek ugy mukodnek, mint a monokrom tintasugaras nyomtatok, de negy tintapatronnal egyszerre. (CMYK) Grafikailag jo eredmenyt adnak es a koltseguk is elfogadhato. (A nyomtato olcso, de a patron nem.)

A legjobb minoseg erdekeben specialis tintara es papirra van szukseg. Ketfele tinta letezik. A **festek alapu tinta** szines elkevert festeket tartalmaz, ami egy folyadek-tartalyban van. Fenyes szine van es konnyen folyik. A fo problemaja, hogy fakulnak a szinek, ha ultraibulya sugar eri oket, mint amilyen a napfenyben van. A **szinezoanyag alapu festek** a szinezoanyag reszecskeit tartalmazza, amit egy folyadektartalyban tarolnak es onnan a papirra illan, hatrahagyva a szinezoanyagot. Eyek nem vesztik el a szinuket, de nem is olyan fenyesek, mint a festek alapu tintak es a szinezoanyag reszecskei hajlamosak eltomiteni a fuvokakat, amit igy rendszeresen takaritani kell. Fenykepek nyomtatasahoz festett, vagy fenyes papirra van szukseg. Ezek a papirok specialisan tervezettek, hogy a festekcseppeket megtartsak es ne follyon szet.

Egy lepessel a tintasugaras nyomtatok felett van a **homogen-tintas nyomtato**. Ezek negy specialis viaszbol keszult teglabol allnak, amiket aztan egy tartalyba olvasztanak. Az ilyen nyomtatok inditasa akar tiz percbe is beletelhet, amig a teglakat megolvasztja. A forro tinta ezutan a papirra kerul, ahol megszilardul es raolvad a papirra mikozben ket kemeny gorgo kozott halad at.

A harmadik fajtaja a szines nyomtatoknak a szines lezernyomtato. Ugy mukodik mint a monokrom testvere, kiveve azt, hogy szetvalasztja a CMYK kepeket, es negy kulonbozo tonert hasznal.

***[106-109]

Oravecz Anikó
T106-109

4 bittel a nyomtatónak 55 MB-ra van szüksége csak a bit-térképhez, nem számítva a memóriát a belső processzorokhoz, fontokhoz stb. Ez az igény teszi a színes lézernyomtatókat drágává, de a nyomtatás gyors, a minőség jó és a képek stabilok maradnak.
A negyedik típusa a színes nyomtatóknak a wax (viasz) nyomtató. Széles szalagja van a négy színű waxnak, ami fel van osztva lap méretű kötegekre.
Fűtő elemek ezrei olvasztják a viaszt, amint a papír alámozdul. A viasz ráolvad a papírra pixelek formájában használva a CMYK rendszert. A wax nyomtatók a legfőbb színes nyomtató technológiát jelentették, mígnem más fajták el nem mozdították erről a helyről, amelyeknek olcsóbb a fogyasztása.
Az ötödik fajtája a színes nyomtatóknak a festék szublimációs nyomtató. Habár ennek vannak Freud-i vonásai, a szublimáció a tudományos neve egy szilárd anyag gázzá alakulásának a folyékony állapot kimaradásával. A száraz jég (fagyasztott carbon dioxid) egy jól ismert anyag, ami szublimál. Egy ilyen nyomtatóban egy "hordozó" tartalmazza a CMYK festékeket, ami kihagyja a meleg nyomtató fejet, ami a programozható fűtő elemeket tartalmazza. A festékek azonnal elpárolognak és a speciális papír felszívja azt. Minden fűtő elem 256 különböző hőfokot képes létrehozni. Minél magasabb a hőmérséklet annál több a festék ami felszívódik, és annál intenzívebb a szín. Eltérő minden más színes nyomtatótól, majdnem folyamatos szín létrehozása lehetséges minden pixellel, így nem szükséges a félárnyékolás. A kis snapshot (pillanatfelvétel) nyomtatók gyakran használják a festékszublimáló eljárást, hogy létrehozzanak egy valóságosabb fotó-képet speciális (és drága) papíron.

2.4.5. Modemek

Az elmúlt évek számítógép-használatának növekedésével általános hogy egy számítógépnek szükséges kommunikálni egy másikkal. Pl. sok embernek van otthon személyi számítógépe, amit arra használ, hogy kommunikáljon a munkahelyi számítógéppel, egy Internet Szolgáltatón keresztül vagy egy házibank rendszerrel. Mindezek az alkalmazások a telefont használják a kommunikációs kapcsolat megvalósításához. Bár egy sima telefonvonal nem alkalmas számítógépes jelek továbbítására, ami 0 Volt esetén 0-t, 3-tól 5 Voltig 1-et jelent, mint az a Fig. 2-39 (a) ábrán látszik. A kétszintű jelek tekintélyes eltorzítást tesznek lehetővé mikor egy hang-továbbító telefonvonalon továbbítunk, ezért továbbítási hibákhoz vezet. Mindemellett egy tisztán szinusz görbe jel 1000 és 2000 Hz frekvencia között - amit carriernek (hordozónak) hívnak - relatíve kevesebb torzítással továbbítható, és ez a tény kihasznált, mint az alapja a legtöbb telekommunikációs rendszernek.
Amiért a szinusz hullám lüktetése teljesen szabályos, egy tiszta szinusz hullám semmilyen információt sem továbbít. Azonban megváltoztatva az amplitúdót (kilengést), a rezgésszámot vagy a fázist egyesek és nullások sorának továbbítása lesz lehetséges, mint ez a Fig. 2-39. ábrán látható. Ezt a fejlesztést modulációnak hívják. Az amplitúdó modulációban (ld. Fig 2-39. (b)) két különböző feszültségi szint használatos 0-ra és 1-re. Hallgatva a digitális adatátvitelt egy nagyon kis adat-sebességnél erős hang hallható 1-nél és nincs hang 0-nál.
A frekvencia modulációban (ld. Fig 2-39 (c)) a feszültségi szint állandó, azonban a frekvencia más 1-nél és 0-nál. Aki egy frekvencia modulált digitális adatátvitelt

hallgat, két hangot hall 0-nak illetve 1-nek megfelelőt. A frekvencia modulációra gyakran mint frekvenciaváltóra utalnak.
Egy egyszerű fázis modulációban (ld. Fig 2-39 (d)) az amplitúdó és a frekvencia nem változik, azonban a fázishordozó 180°-kal elfordul amikor 0-ról 1-re vagy 1-ről 0-ra vált az adat. A kifinomultabb fázis modulált rendszerekben minden oszthatatlan időköz kezdetén a hordozó fázisa hirtelen vált 45, 135, 225 vagy 315°-ot időközönként 2 bitet megengedve. Ezt dibit fázisködolásnak hívják. Pl. egy 45°-os fázisváltás 00-t jelenthet, egy 135°-os 01-et, és így tovább. Más tervek is léteznek időközönként 3 vagy több bit továbbítására időközönként. Az időközök száma (a lehetséges jelváltások száma másodpercenként) a "baud" mérték. 2 vagy több bittel időintervallumonként a bit arány meghaladja a "baud" mértékét. Sok ember összetéveszti ezt a két időtartamot.
Ha a továbbított adat 8 bites karakterek sorozatát tartalmazza, kívánatos lenne egy 8 bit szimultán (azonos időbeli) továbbítására képes kapcsolat - ami 8 pár vezeték. Amiért a hang-továbbítású telefonvonalak csak egy csatornáról gondoskodnak a biteket sorban kell küldeni, egyiket a másik után (vagy kettes csoportokban ha dibit kódolót használunk).
Az eszköz, amely egy számítógéptől karaktereket fogad kétszintű jelek formájában, egy bitet egyszerre, és továbbítja a biteket egyes vagy kettes csoportokban amplitúdó, frekvencia vagy fázis modulált formában, a modem. Minden karakter elejének és végének jelzésére egy 8 bites karakter egy megelőző kezdet bittel és egy azt követő stop bittel küldődik el, 10 bitet készítve mindből.
A továbbító modem sajátos biteket küld egy karakteren belül szabályosan osztott időközönként. Pl. 9600 baud magába foglal egy jelváltozást minden 104 µsec-ban. Egy másik modem a fogadás végén átalakította a megváltoztatott hordozót bináris számmá. Amiért a bitek a fogadóra szabályosan osztott időközönként érkeznek egyszer a fogadó modem megállapította a kezdetét a karakternek, és az órája mondta meg, mikor próbálja olvasni a vonalakon érkező biteket.
A modern modemek 28800 bit/sec és 57600 bit/sec közötti adatgyorsasággal működnek. Ezek különböző technikákat használnak több bit küldésére baud-onként, modulálva az amplitúdót, a frekvenciát és a fázist. Majdnem mind ezek közül full duplex (teljes dupla), ami azt jelenti, hogy mindkét irányban tud továbbítani egy időben (különböző frekvenciákat használva). A modemeket vagy továbbító vonalakat, amik csak egy irányban tudnak egyszerre továbbítani (mint egy "egysávos" vasút, amely kezeli az észak felé tartó, illetve a dél felé tartó vonatokat, de nem ugyanabban az időben) half duplexnek (fél dupla) hívják. Azok a vonalak, melyek csak egy irányba továbbítanak simplexnek (egyszeres) hívják.
Figure 2-39. A 01001011000100 bináris szám továbbítása telefonvonalon keresztül bitről bitre. (a) Kétszintű jel. (b) Amplitúdó moduláció. (c) Frekvencia moduláció. (d) Fázis moduláció.

ISDN

A korai 1980-as években az Európai PTTs (Posta, Telefon és Telegráf Adminisztrációk) feltalálták egy szabványt a digitális telefonokhoz, amit ISDN-nek (Integrált Szolgáltatások Digitális Hálózata) neveznek. Ezt arra tervezték, hogy lehetővé tegye a nyugdíjasoknak, hogy olyan riasztójuk legyen a házukban, amely egy központi megfigyelő/vészjelző rendszerhez kapcsolódik és sok más érdekes, addig nem létező alkalmazást. És akkor hirtelen megjelent a World Wide Web és az emberek nagyobb sávszélességű digitális Internetelérést követeltek. Bingo. Az ISDN azonnal felfedezte "gyilkos" alkalmazását (bár tervezői hibáján kívül). Híressé vált

ugyanúgy az Egyesült Államokban mint máshol.
Mikor egy telco (ipari zsargon a telefonos társaságokra) ügyfél előfizet az ISDN-re, a telco lecseréli a régi analóg vonalat egy digitálisra. (Valójában a vonalat magát nem cserélik ki, csak a berendezést mindkét végén.) Az új vonal két független digitális csatornát tartalmaz mindkettőn 64000 bit/sec-mal, és még egy jeladó csatornával 16000 bit/sec-mal. A berendezés a 3 csatornát kombinálja egy egyszerű 144000 bit/sec-os digitális csatornává. Az üzletelésre egy 30 csatornás ISDN vonal érhető el.
Nem csak az ISDN gyorsabb az analóg csatornánál, de ez lehetővé teszi, hogy általánosan nem több mint 1 sec alatt kapcsolat létesüljön, nem hosszabbat kívánva meg egy analóg modemtől, és az sokkal megbízhatóbb (kevesebb hibával) mint az analóg vonalak. Választéka van még az extra tulajdonságoknak és opcióknak is, amelyek nem mindig érhetők el analóg vonalakkal.
Egy ISDN kapcsolat felépítése látható a Fig 2-40-es ábrán. Amit a hordozó biztosít egy digitális bit cső, ami csak mozgatja a biteket. Amit a bitek jelentenek, fenn van a küldőn és a fogadón. Az interface az ügyfél berendezése és a hordozó berendezése között az NT1 készülék T interface-szel az egyik oldalon és U-val a másikon. Az U.S.-ban az ügyfeleknek meg kell vásárolni a saját NT1 készüléküket. A legtöbb európai országban ezt kölcsönözni kell a hordozótól.

2.4.6. Karakter kódok

Minden számítógépnek van egy karakter-beállítása, amit használ. Minimumként ez a beállítás a 26 nagybetűt, a 26 kisbetűt, a számokat 0-tól 9-ig és a speciális szimbólumok beállításait - mint szóköz, pont, mínuszjel, vessző - tartalmazza.
Ezen karakterek számítógépbe helyezéséhez mind egy számmal van jelölve: pl. a=1, b=2,…,z=26, +=27, – =28. A karakterek feltérképezését egész számokra karakter kódnak hívják. Alapvető, hogy a kommunikáló számítógépek ugyanazt a kódot használják, vagy máskülönben nem lennének képesek megérteni egymást. Ebből az okból standardokat fejlesztettek ki. Lejjebb tanulmányozni fogunk kettőt a legfontosabbak közül.

ASCII

Az egyik legszélesebb körben használt kód az ASCII (American Standard Code for Information Interchange). Minden ASCII karakter 7 bites, 128 karaktert engedve meg. Fig 2-41 mutatja az ASCII kódot. A kódok 0-tól 1F-ig (16-os számrendszer) karaktervezérlők és nem nyomtatottak.
Sokat az ASCII vezérlő karakterek közül adat továbbításra terveztek. Pl.: egy üzenet állhat egy SOH (Start of Header) karakter, egy fejes, egy STX (Start of Text), a szöveg maga, ETX (End of Text)

***[110-113]

110-113old.

A gyakorlatban azonban az üzenetek formátuma telefonon és a hálózatokon küldve teljesen más. Itt az ASCII átvitel karaktereket nem sokat használják.

Az ASCII nyomtató karakterek egyértelműek. Magukban foglalják a nagy és kis betűket, a számjegyeket, az írásjeleket és néhány matematikai szimbólumot.

2.4. UNICODE

A számítógépipar az USA-ban fejlődött ki, ez határozta meg az ASCII karakter táblát. Az ASCII az angol nyelvnek megfelel, de más nyelveknek kevésbé. A franciához ékezetek kellenek (pl.:systéme); a némethez diakritikus jelek (pl.: für) és így tovább. Néhány európai nyelvben vannak olyan betűk melyek az ASCII-ben nem találhatók meg, mint például a német ß. Vannak nyelvek amelyeknek abc-je teljesen más (pl.: orosz, arab), és vannak nyelvek amelyeknek egyáltalán nincs abc-je (pl.: kínai). Ahogy a számítógép az egész földön kezdett elterjedni, a szoftver árusok termékeiket olyan országokban is el akarják adni, ahol a használók nem beszélnek angolul, ezért egy más karakter táblára volt szükség.

Az első kísérlet az ASCII kibővítésére az IS 646 volt, amely újabb 128 karaktert adott az ASCII-hez, 8-bites kóddá alakítva Latin-1 néven. A hozzáadott karakterek főleg latin betűk voltak ékezetekkel és diakritikus jelekkel. A következő próbálkozás az IS 8859 volt, ami bevezette a kód lap fogalmát, ami nem más mint 256 karakteres készlet bizonyos nyelvek vagy nyelvcsoportok részére. Az IS 8859-1 a Latin-1. IS 8859-2 a latin alapú szláv nyelvekkel foglalkozik (pl.: cseh, lengyel, magyar). AZ IS 8859-3 már olyan karaktereket is tartalmaz, amelyeket többek között a török, a máltai, az eszperantó és a galíciai nyelv használ. A probléma a kódlap beállításával az, hogy a szoftvernek számon kell tartania, hogy melyik lapot használja, ezenkívül lehetetlen keverni a nyelveket a lapokon keresztül, valamint a táblázatok egyáltalán nem fedik a japán és kínai nyelvet.

A számítógép gyártók egy csoportja elhatározta, hogy megoldják a problémát azzal, hogy konzorciumot hoznak létre egy új rendszer megformálásához UNICODE néven és ezt terjesztik nemzetközi szabványként (IS 10646). Az UNICODE-ot támogatja néhány programozási nyelv (pl.: Java), néhány operációs rendszer (pl.: Windows NT) és más alkalmazások. Úgy tűnik, hogy egyre elfogadottabb lesz, ahogy a számítógépipar globálissá válik. Az UNICODE alapötlete, hogy minden karakterhez és szimbólumhoz egyedi állandó 16-bites értéket rendel, melyet kódpontnak hívnak. Nincsenek több bájtos karakterek és nem használhatunk menekülési szekvenciákat. Mivel minden szimbólum 16.bites, így a szoftver írás sokkal egyszerűbb.

A 16-bites szimbólumokkal az UNICODE-nak 65,536 kódpontja van. A világnyelvek együttesen körülbelül 200,000 szimbólumot használnak, és mivel a kódpont kevés, ezért a kiosztását nagyon körültekintően kell elvégezni. A kódpontok közel felét már kiosztották. Az konzorcium figyeli a javaslatokat, hogy feldolgozza a többit. Hogy felgyorsítsák az UNICODE elterjeszteset, a konzorcium okosan a Latin-1-et használta a 0-tól 255-ig terjedő kódpontokhoz, megkönnyítve ezzel az ASCII-ről az UNICODE-ra való átállást.

Hogy ne veszítsenek kódpontot, minden diakritikus jelnek meg van a saját kódpontja, és a szoftver dolga, hogy azt a szomszédos jellel összekapcsolva egy új

karaktert alkosson.

A kódpont teret blokkokra osztják, melyek mindegyike 16 kódpontot tartalmaz. Az UNICODE fő abc-i egymást követő blokkokban helyezkednek el.

Néhány példa (és a kódpontok számának megállapítása) a latin (336), a görög (114), a cirill (256), az örmény (96), a héber (112), a devanagari (128), a gurmukhi (128), az oriya (128), a telugu (128) és a kannada (128). Jegyezzük meg, hogy mindegyik ezek közül a nyelvek közül több kódpontot határozott meg, mint ahány betűje van. Ez a választás részben azért történt, mert néhány nyelvben mindegyik betűnek több alakja van. Például az angolban minden betűnek két alakja létezik: kis betű és nagy betű. Néhány nyelvben három vagy több alak van, valószínűleg attól függően, hogy a szó elején, közepén vagy végén van.

Ráadásul ezek az abc-k mellé még kódpontokat határoznak meg diakritikus jeleknek (112), írásjeleknek (112), alsó és felső indexnek (48), pénznem szimbólumoknak (48), matematikai szimbólumoknak (256), geometriai formáknak (96) és dingbatoknak (192).

Ezek után jönnek a szükséges szimbólumok a kínaihoz, japánhoz és koreaihoz. Először 1024 fonetikus szimbólum (pl. katakana és bopomofo) és utána a kínaiban és japánban használt egybefoglalt Han ideogrammák (20992) és a koreai Hangul szótagok (11156).

Hogy a használók különleges karaktereket használjanak különleges szándékokra, 6400 kódpontot határoztak meg helyi használatra.

Amíg az UNICODE nemzetköziséggel kapcsolatos problémákat old meg, nem oldja meg (próbálja megoldani) a világ összes problémáját. Például míg a latin abc sorban van, a Han ideogrammák nincsenek abc sorrendben, ennek egy következménye, hogy egy angol program meg tudja vizsgálni a “macska” és “kutya” szót és sorrend szerint csoportosítani egyszerűen az első betűjük UNICODE értékének összehasonlításával, de egy japán programnak külső táblázatra van szüksége, hogy rájöjjön a két szimbólum közül melyik van előbb a szótárban.

Egy másik vitapont, hogy minden pillanatban új szavak bukkannak fel. 50 évvel ezelőtt senki se beszélt appletekről, kibertérről, gigabájtról, lézerről, modemről vagy videokazettáról. Az angolban az új szavak hozzáadása nem követel új kódpontokat, de a japán igen. Ráadásul az új műszaki szavak legalább 20,000 új (főleg kínai) személy és hely nevet igényelnek. A vakok szerint Braille-írásnak (vakírásnak) is lennie kellene benne, és mindenféle speciális érdeklődésű csoportok azt akarják, amit megértenek, mint jogos kódpontjukat. Az UNICODE konzorcium átnézi és dönt minden új javaslatról.

Az UNICODE ugyanazokat a kódpontokat használja azokhoz a karakterekhez amelyek majdnem ugyanúgy néznek ki, de más jelentésük van vagy japánul és kínaiul egy kicsit másképp írják (mint ahogy az angol szófeldolgozók a “blue” szót ugyanúgy betűzik, mint a “blew”-t, mivel a kiejtésük azonos). Néhányan ezt a kevés kódpont miatti optimalizálásnak tartják, mások Angol-szász imperializmusnak (és még azt gondolják, hogy 16-bites értéket rendelni egy karakterhez nem politika volt?). Hogy súlyosbítsuk a bajokat a teljes japán szótárban 50000 kanji van (a nevek kivételével), és csak 20992 lehetséges kódpont a Han ideogrammákhoz. Választani kell, és nem minden japán véli úgy, hogy a komputer cégek konzorciuma, mégha közülük páran japánok, az ideális fórum ezekhez a döntésekhez.

2.5. Összefoglalás

A komputerrendszerek 3 típusú komponensből állnak: processzorból,

memóriából és I/O készülékekből. A processzor feladata, hogy egyszerre egy utasítást vegyen a memóriából, dekódolja és végrehajtsa azt. A szállító- dekódoló- végrehajtó kör algoritmusként is leírható, sőt néha egy alacsonyabb fokon irányító szoftver tolmács fejleszti ki. Hogy sebességet nyerjen, néhány komputernek egynél több vezetéke van, vagy szuperskalár tervezésű többfunkciós egységekkel, melyek párhuzamosan működnek.

A többprocesszoros rendszerek egyre gyakoribbak. Párhuzamos komputerekben sorprocesszorok vannak beépítve, amelyben ugyanazt a működést ugyanabban az időben végrehajtják. Multiprocesszor, melynél több CPU osztozik egy közös memórián. Multikomputer, ahol több komputernek megvan a saját memóriája, de üzenetek küldésével kommunikálnak.

A memóriát elsődlegesként vagy másodlagosként kategorizálhatjuk.

Az elsődleges memóriát a pillanatnyi program működésére használjuk. Az elérési ideje rövid, maximum néhány 10 nanoszekundum és független a cím hozzáférhetőségétől. A cache ezt az időt még jobban lecsökkenti. Néhány memória hibajavító kódokkal van ellátva, hogy a megbízhatóságot növeljék.

A másodlagos memóriáknak azonban sokkal hosszabb az elérési ideje (milliszekundum vagy több) és az olvasott vagy írt adatok helyétől függ. Mágnesszalag, mágneslemez és optikai lemez a leggyakoribb másodlagos memóriák. A mágneslemezek sok variációban léteznek, beleértve a floppy lemezt, a winchestert, az IDE és SCSI lemezt, valamint a RAID-t is. Optikai lemezek a CD-ROM a CD-R és DVD.

I/O eszközöket az információ átvitelére használják. Egy vagy több buszon csatlakozik a processzor(ok)hoz és a memóriához. Például a terminálok, egerek, nyomtatók és modemek. A legtöbb I/O eszköz az ASCII kódkészletet használja, vár az UNICODE elfogadása, ahogy a komputeripar globálissá válik.

Feladatok

1. Vegyük a gép működését a 2.2 kép alapján. Tegyük fel, hogy az ALU regiszterbe a betöltés 5 nanoszekundumig tart, az ALU 10 nanoszekundumig fut, majd az eredmény tárolása 5 nanoszekundumot vesz igénybe. Mennyi a legnagyobb MIPS, amit ez a gép elérhet csővezeték nélkül?
2. Mi a szándéka a Sec. 2.1.2-es listán a 2. Lépésnek? Mi történne, ha kihagynánk ezt a lépést?
3. Az 1-es számítógép minden utasítást 10 nanoszekundum alatt végez el. A 2-es számítógépnek mind 5 nanoszekundumig tart. Biztosan állíthatjuk, hogy a 2-es számítógép gyorsabb? Fejtsd ki!

***[114-117]

4. Tegyük fel, hogy egy egy-chipes számítógépet tervezel beágyazott rendszerekhez. A chip tartalmazza saját memóriáját, és a CPU-val egy sebességgel működik, elérési büntetés nélkül. Vizsgáld meg a 2.1.4-es alfejezetben megvitatott elveket, és magyarázd meg, vajon miért ilyen fontosak még mindig (feltételezve, hogy a magas teljesítmény még mindig követelmény).

5. Lehetséges lenne a 2-8(b) ábrán szereplő processzor gyorsítása? Ha igen, mi az első megoldandó probléma?

6. Néhány számítás magasan szekvenciált - ezeknél minden lépés függ az őt megelőzőtől. Ezekhez a számításokhoz egy array, vagy egy pipeline processzor lenne megfelelőbb? Miért?

7. Hogy versenyezni tudjanak az újonnan felfedezett nyomtatási eljárással, egy középkori kolostor eldöntötte, hogy kézzel írott, papírborítású könyvek tömegtermeléséhez kezd nagyszámú írnok egyidejű alkalmazásával. Mikor az apát felolvassa a készülő könyv első szavát, az összes írnok leírja ezt. Ez a folyamat addig folytatódik, míg a teljes könyvet fel nem olvasták és le nem másolták. A 2.1.6-os alfejezetben tárgyalt párhuzamos feldolgozó rendszerek közül melyik hasonlít leginkább erre a módszerre?

8. Ahogy lefelé haladunk a szövegben megtárgyalt ötszintű memória-hierarchiában, az elérési idő növekszik. Készíts ésszerű becslést az elérési idő arányáról optikai lemez és regiszter memória között. Tételezzük fel, hogy a lemez már csatlakoztatva van.

9. Számold ki az emberi szem által használt adatmennyiséget a következő információk alapján. A látómező körülbelül 1 millió (10^6) elemet (képkockát) tartalmaz. Mindegyik képkockát le lehet egyszerűsíteni a három elsődleges szín kombinációjára, mindegyik színt 64 árnyalattal számítva. Az időbeli felbontás 100 msec.

10. Az élő szervezetek genetikai információi a DNS molekulákban vannak kódolva. A DNS molekula a négy alapvető nukleotida lineáris szekvenciája: az A-é, a C-é, a G-é és a T-é. Az emberi génstruktúra körülbelül 3 milliárd (3*10^9) nukleotidát tartalmaz körülbelül 100,000 gén formájában. Mennyi információt tárol az emberi génstruktúra (bitekben)? Mekkora az információkapacitása egy átlagos génnek (bitekben)?

11. A következő memóriák közül melyek lehetségesek? Melyek észszerűek? Miért?
a. 10 bites cím, 1024 cella, 8 bites cellaméret
b. 10 bites cím, 1024 cella, 12 bites cellaméret
c. 9 bites cím, 1024 cella, 10 bites cellaméret
d. 11 bites cím, 1024 cella, 10 bites cellaméret
e. 10 bites cím, 10 cella, 1024 bites cellaméret
f. 1024 bites cím, 10 cella, 10 bites cellaméret

12. A szociológusok három lehetséges választ ismernek egy tipikus

közvéleménykutatási kérdésre, mint az "Ön hisz a fogtündérekben?" -név szerint: igen, nem és tartózkodom. Ennek tudatában a Szociomagnetikai Számítástechnikai Társaság eldöntötte, hogy épít egy olyan számítógépet, ami képes feldolgozni ezeket az adatokat. A számítógépnek trináris memóriája volt - azaz, minden byte (tryte) 8 tritet tartalmazott, ahol minden trit a 0, 1, 2 értékeket veheti fel. Mennyi trit szükséges egy 6 bites szám tárolásához? Adj egy kifejezést arra, hogy n bit tárolására mennyi tritre van szükség!

13. Egy számítógép fel lehet szerelve 268,435,456 byte-nyi memóriával. A gyártók miért ilyen sajátos számokat választottak, a könnyen megjegyezhető számok helyett, mint pl: 250,000,000?

14. Találj ki egy páros paritású Hamming kódot a 0-tól 9-ig való számokra!

15. Találj ki egy olyan kódot a 0-tól 9-ig való számokhoz, aminek Hamming-távolsága 2!

16. Hamming kódban néhány bitet "elpazarlunk", abban az értelemben, hogy ellenőrzésre használjuk információtárolás helyett. Egy 2^n-1 hosszú (adat+ellenőrzés) üzenet hány százaléka ilyen "elpazarolt" bit? Becsüld meg ennek a kifejezésnek az értékét, az *n* 3-tól 10-ig tartó intervallumán!

17. A telefonvonalak átviteli hibái gyakrabban fordulnak elő halmozottan (egymást követő bitek eltorzulnak), mint egyesével. Mivel az alap Hamming kód karakterenként csak egy hibát képes kijavítani, használhatatlan, ha egy adott zavarás *n* db egymás utánni bitet torzít el. Találj ki ASCII karakterek vonalon való továbbításának egy olyan módszerét, ahol a zavarások eltorzíthatnak 100 egymást követő bitet. Feltételezve, hogy a minimális távolságkét zaj között 1000 karakter. *Ötlet:* Gondolkodj el figyelmesen a bitátvitel sorrendjéről.

18. Mennyi ideig tart elolvasni egy 800 cilinderes lemezt? Minden cilinder öt sávot tartalmaz, és minden sáv 32 szektort. Először is, az olvasás a 0. sáv 0. szektorából indul, ezután az 1. sáv 0. szektorából, és így tovább. A forgási idő 20 msec, és a pozícionálás szomszédos cilinderek között 10 msec-ig, legrosszabb esetben 50 msec-ig tart. Egy cilinder sávjai között a váltás azonnali.

19. A 2-19-es ábrán látható lemez 64 sávot/szektort tartalmaz, fordulatszáma 7200 fordulat/perc. Mennyi a lemez elviselhető átviteli aránya egy sávon keresztül?

20. Egy számítógépnek 25 nsec ciklusidejű busza van, ami alatt ír, vagy olvas egy 32 bites szót a memóriából. A számítógépnek van egy Ultra-SCSI lemeze, ami a buszt használja, és 40 MByte/sec sebességgel fut. A CPU normálisan 25 nsec idő alatt szállít és hajt végre egy utasítást. Mennyire lassítja le ez a lemez a CPU-t?

21. Képzeld el, hogy egy operációs rendszer lemezkezelő részét írod. A lemezt, logikusan, blokkok sorozataként ábrázolod, a belül lévő 0-tól, a kivül lévő maximumig. Ahogy egy file elkészül, meg kell határoznod az üres szektorokat. Ezt megteheted belülről kifelé, vagy kívülről befelé. Számít valamit, melyik stratégiát választod? Indokold válaszod!

22. Az LBA címzés 24 bitet használ egy szektor megcímzésére. Mekkora az a legnagyobb lemez, amit ezzel még kezelni lehet?

23. A 3-as szintű RAID az egybites hibákat képes egy paritás meghajtó használatával javítani. Mi az értelme a 2-es szintű RAID-nek? Mindezek után, ez csak egy hibát bír kijavítani és több meghajtót használ erre.

24. Mennyi a pontos adatkapacitása (byte-okban) egy szabványos, 74 percnyi adatot tartalmazó 2-es módú CD-ROM-nak?

25. Egy CD-R beégetéséhez a lézernek nagyon gyorsan kell ki-be pulzálnia. Amikor 4x-es sebességgel fut 1-es módban, mekkora az impulzusok hossza nanoszekundumokban?

26. Ahhoz, hogy elhelyezzünk egy 133 perces videót egy egyoldalas, egyrétegű DVD-n, egy tisztességes tömörítési mennyiség szükséges. Számold ki a szükséges tömörítési arányt! Feltételezve, hogy 3,5 GB hely szabad a videó sávnak, a képfelbontás 720x480 pixel 24 bit színnel, és a képlejátszás 30 képkocka/másodperc.

27. Az átviteli arány a CPU és társított memóriája között nagyságrendekkel nagyobb, mint a mechanikus I/O átviteli arány. Hogyan okozhat ez az egyensúlytalanság hatékonytalanságot? Hogyan lehet ezt enyhíteni?

28. Egy bit-térkép terminál felbontása 1024x768. A képernyőt egy perc alatt 75-ször rajzolja újra. Mennyi az impulzus egy képpontra átszámítva?

29. Egy gyártó olyan színes, bit-térkép terminált hirdet, ami 2^24 különböző színt képes megjeleníteni. Ennek ellenére a hardware csak egy byte-ot használ minden pixelhez. Hogyan lehetséges ez?

30. Egy bizonyos karakterkészletben egy monokróm lézer nyomtató 50, 80 karakteres, sort tud nyomtatni oldalanként. Egy átlagos karakter egy 2mm x 2mm-es dobozt foglal el, aminek körülbelöl 25%-a festék. A többi üres. A festékréteg 25 micron vastag. A nyomtató festékpatronja 25cm x 8cm x 2cm. Hány oldal kinyomtatására elegendő egy festékpatron?

31. Ha egy páros paritású ASCII szöveg aszinkron átvitele 2880 karakter/sec egy 28,800 bps-es modemen keresztül. A kapott bitek hány százaléka tartalmaz adatot (a hallottakkal ellentétben)?

32. A Hi-Fi Modem Társaság elkészített egy új frekvencia-modulációs modemet, ami 16 frekvenciát használ 2 helyett. Minden perc *n* egyenlő időintervallumra van felosztva, mindegyik a 16 hang egyikét tartalmazza. Szinkron átvitelt használva hány bit/sec ennek a modemnek az átviteli sebessége?

33. Becsüld meg, hány karaktert, beleértve a szóközt is, tartalmaz egy tipikus számítástechnikai szakkönyv! Hány bit szükséges egy ilyen könyv lekódolásához ASCII-ben, ellenőrzéssel? Mennyi CD-ROM szükséges egy 10,000 könyvből álló

számítástechnikai szakszótár tárolásához? Mennyi duplaoldalú, duplarétegű DVD szükséges ugyanezen könyvtár tárolásához?

34. Dekódold a következő bináris ASCII szövegeket: 1001001, 0100000, 1001100, 1001111, 1010110, 1000101, 0100000, 1011001, 1001111, 1010101, 0101110.

35. Írj eljárást *hamming(ascii, kódolt)* néven, ami az *ascii* alacsony helyiértékű 7 bitjét egy 11 bites egész *kódolt*-ban tárolt kódszóba kódolja.

36. Írj függvényt *távolság(kód, n, k)* néven, ami kap egy *kód* nevű tömböt, ami *n* db *k* bites karakterből áll, bemenetként, és a karakterhalmaz távolságával tér vissza kimenetként.

3

A digitális logika szintje

Az 1-2-es ábra hierarchiájának legalján találhatjuk a digitális logika szintjét, a számítógép igazi hardverét. Ebben a fejezetben több szempontból is megvizsgáljuk a digitális logikát, mint az elkövetkezendő fejezetek megértésének alapkövét. Ez a tárgy a számítástudomány és az elektroműszerészet határán helyezkedik el, de önmagában is érthető, így nem szükséges megértéséhez semmiféle előzetes hardver, vagy mérnöki tapasztalat.

Az alapelemek, melyekből a digitális számítógépek felépülnek meglehetősen egyszerűek. Tanulmányainkat ezek az alapelemek , és az analizálásukra használt speciális, kétváltozós algebra (Boolean algebra) megismerésével kezdjük. Ezután megvizsgálunk néhány alapvető áramkört, amit kapuk egyszerű kombinációiból kaphatunk, beleértve az aritmetikai áramköröket is. A következő téma az, hogyan használhatunk kapukat információ tárolására, azaz, a memóriák felépítése. Ezután a CPU-hoz érünk, különösen az egychipes CPU-k kapcsolódásaihoz a memóriával, és a perifériákkal. A fejezet végén számos példa található az iparból.

3.1 Kapuk és Boolean algebra

Digitális áramköröket néhány primitiv elem számtalan kombinációjából készíthetünk. A következő alfejezetben leírjuk ezeket a primitív elemeket, megmutatjuk, hogyan lehet őket kombinálni, és bevezetünk egy hatásos matematikai teknikát, amivel a viselkedésüket lehet vizsgálni.

***[118-121]

Hidasi László
h938426@stud.u-szeged.hu

3.1.1 Kapuk

A digitális áramkörben csak két logikai érték létezik. Tipikusan a 0 és 1 volt közötti jel fejez ki egy értéket (pl. a kettes számrendszerbeli 0-t), és a 2 és 5 volt közötti jel egy másikat (pl. a kettes számrendszerbeli 1-et). Az e két intervallumon kívül eső feszültségek nem megengedettek. Kis elektronikus eszközök, **kapuk** számítják e kétértékű jelek különböző függvényeit. Ezek a kapuk alkotják azt a hardver bázist, amelyre az összes digitális számítógép épül.

A kapuk belső működési részletei nem tartoznak a könyv anyagához, mert az **eszközök szintjéhez** tartoznak, amely a 0. szint alatt van. Mindazonáltal most eltérünk a tárgytól, hogy egy pillantást vessünk az alapötletre, ami nem bonyolult. Az összes modern digitális logika végül is azon a tényen alapszik, hogy egy tranzisztor úgy is működhet, mint egy nagyon gyors kettes számrendszerbeli kapcsoló. A 3-1(a) ábrán egy kétpólusú tranzisztor (a kör) egy egyszerű áramkörbe beágyazva látható. Ennek a tranzisztornak három kapcsolata van a külvilággal: a **bemenet,** az **alap,** és a **feszültség-levezető.** Amikor a bemeneti feszültség, V_{be} egy bizonyos kritikus érték alá csökken, a tranzisztor kikapcsol, és végtelen ellenállásként kezd viselkedni. Ez azt okozza, hogy az áramkör kimeneti feszültsége, V_{ki} egy V_{cc} -hez közeli értéket vesz fel, egy kívülről beállított feszültséget, ami tipikusan +5 volt ilyen típusú tranzisztornál. Amikor V_{be} átlépi a kritikus értéket a tranzisztor bekapcsol és drótként kezd el működni, mire V_{ki} földelt lesz (megállapodás szerint 0 volt).

3-1-es ábra. (a) Egy tranzisztorfordító (b) Egy NAND kapu (c) Egy NOR kapu

A fontos dolog akkor történik, amikor V_{be} alacsony, V_{ki} magas, és fordítva. Ez az áramkör tehát egy fordító, a logikai 0-t logikai 1-é, a logikai 1-et logikai 0-vá alakítja. Az ellenállásra (a szaggatott vonal) azért van szükség, hogy korlátozza a tranzisztor áramfelvételét, így nem fog kiégni. A szükséges idő arra, hogy az egyik állásból a másikba kapcsoljon, általában néhány nanoszekundum.

A 3-1 (b) ábrán két tranzisztor van sorba kapcsolva. Ha V_1 és V_2 is magas, mindkét tranzisztor vezetni fogja az elektromosságot, és V_{ki} alacsony lesz. Ha valamelyik bemenet alacsony, a megfelelő tranzisztor kikapcsol, és a kimenet magas lesz. Más szóval V_{ki} akkor és csak akkor lesz alacsony, ha V_1 és V_2 is magas.

A 3-1 (c) ábrán a két tranzisztor nem sorba, hanem párhuzamosan van összekapcsolva. Ennél az állásnál ha valamelyik bemenet magas, a megfelelő tranzisztor bekapcsol és a kimenet földelt lesz. Ha mindkét bemenet alacsony, a kimenet magas marad.

Ez a három áramkör, vagy a velük egyenértékűek a legegyszerűbb áramkörök. A nevük NOT, NAND és NOR kapuk, egyenként. A NOT kapukat **fordító**knak is gyakran hívják, a két formát váltakozva fogjuk használni. Ha alkalmazzuk azt a megállapodást, miszerint “magas” (V_{cc} volt) a logikai 1, és “alacsony” (földelt) a logikai 0, akkor kifejezhetjük a kimeneti értéket a bemeneti értékek függvényeként. A jelek, amelyeket e három áramkör ábrázolására használnak az áramkörök működés

közbeni magatartásával , a 3-2 (a)-(c) ábrákon láthatóak. Ezeken az ábrákon A és B a bemenet és X a kimenet. Minden sor a kimenetet határozza meg a bemenetek különböző kombinációi esetén.

3-2-es ábra Az öt alapkapu jele és működése

Ha a 3-1 (b) kimeneti jelét egy fordítókörön vezetjük át, egy újabb áramkört kapunk, ami a NAND kapu pontos ellentéte - mégpedig egy áramkör, melynek kimenete akkor és csak akkor 1, ha mindkét bemenet 1. Ennek az áramkörnek neve AND (és) kapu, a jele és a működési leírása a 3-2 (d) ábrán látható. Hasonlóan a NOR kapu is csatlakoztatható egy fordítóhoz, ezzel egy olyan áramkört kapunk, amelynek kimenete 1, ha valamelyik vagy mindkét bemenet 1, de 0, ha mindkét bemenet 0. A jele és a működési leírása ennek az áramkörnek, amelynek neve OR (vagy) kapu a 3-2 (e) ábrán látható. A kis körök, amelyek a fordító, a NAND és a NOR kapu részeként láthatóak az ábrákon, a fordítóbuborékok. Más szövegekben is használják őket a fordított jel jelölésére.

A 3-2 ábrán látható öt kapu a digitális logika szintjének legfontosabb építőeleme. A megelőzőkből világos, hogy a NAND és a NOR kapukhoz két-két tranzisztor szükséges, míg az AND és OR kapukhoz három-három. Emiatt sok számítógép NAND és NOR kapukon alapszik az ismertebb AND és OR kapuk helyett. (A gyakorlatban a kapuk egy kissé máshogy működnek, de a NAND és NOR kapuk még mindig egyszerűbbek, mint az AND és OR kapuk.) Futólagosan érdemes megemlíteni, hogy a kapuknak lehet kettőnél több bemenetük. Általában egy kapunak, például egy NAND kapunak korlátlan számú bemenete lehet, de a gyakorlatban a nyolcnál több bemenet szokatlan.

Bár a kapuk felépítése az eszközök szintjéhez tartozik, meg szeretnénk említeni a gyártási technológiák nagy családjait, mert sokat utalnak rájuk. A két nagy technológia a **kétpólusú** és a **MOS** (Metal Oxide Semiconductor, vas-oxid félvezető). A két nagy kétpólusú típus a **TTL** (Transistor-Transistor Logic, tranzisztor-tranzisztor logika), amely évekig volt a digitális elektronika "igáslova" és **ECL** (Emitter-Coupled Logic), melyet akkor használnak, amikor nagy sebességű működésre van szükség.

A MOS kapuk lassabbak, mint a TTL és az ELC, de sokkal kevesebb áramot igényelnek, és sokkal kisebb helyet foglalnak, tehát sokat egymás mellé lehet helyezni belőlük szorosan. A MOS több variációban létezik, ezek a PMOS, NMOS és CMOS. Bár a MOS tranzisztorok másképpen épülnek fel, mint a kétpólusú tranzisztorok, az a képességük, hogy elektronikus kapcsolóként működjenek, azonos. A legtöbb modern CPU és memória a CMOS technológiát alkalmazza, amely +3,3 volton működik. Ez minden, amit mondani fogunk az eszközök szintjéről. Azok az olvasók, akik tanulmányozni szeretnék ezt a szintet, nézzék meg a 9. fejezetben ajánlott olvasmányokat.

3.1.2 Boole algebra

A kapuk kombinációjából felépített áramkörök leírására egy új típusú algebrára van szükség, egy olyanra, amelyben a változók és a függvények csak a 0 és 1 értékeket vehetik fel. Az ilyen algebra neve **Boole algebra,** amit a feltalálójáról, George Boole (1815-1864) angol matematikusról neveztek el. Az igazat megvallva, valójában a Boole algebra egy speciális típusára utalunk, a **kapcsoló algebrára**, de a "Boole

algebra” kifejezés olyan széles körökben jelentett “kapcsoló algebrát”, hogy nem teszünk a kettő között különbséget.

Ahogy vannak függvények a “rendes” (azaz középiskolai) algebrában, úgy vannak függvények a Boole algebrában. A Boole függvénynek egy vagy több bemeneti változója van, és a kapott eredmény csak e változók értékeitől függ. Egy egyszerű függvény, *f* meghatározható úgy, hogy azt mondjuk, hogy ha *f*(A) akkor 1, ha A 0, és *f*(A) akkor 0, ha A 1. Ez a függvény a NOT (nem) függvény a 3-2 (a) ábrán.

Mivel egy *n* változós Boole függvény bemeneti értékeinek csak 2^n lehetséges kombinációja van, a függvény teljesen leírható egy 2^n soros táblázattal, amelyben minden sorban a függvény értéke található a bemeneti értékek különböző kombinációja esetén. Az ilyen táblázat az **igazságtáblázat.** A 3-2-es ábra táblázatai mind példák igazságtáblázatokra. Ha megállapodunk abban, hogy az igazságtáblázat sorait mindig növekvő sorrendben írjuk ki (kettes számrendszerben), akkor két változóra a sorrend 00, 01, 10 és 11, a függvény teljesen leírható a 2^n bites kettes számrendszerbeli számmal, amit úgy kapunk, hogy az igazságtáblázat eredményoszlopát függőlegesen olvassuk. Így NAND 1110, NOR 1000, AND 0001 és OR 0111 lesz. Nyilvánvalóan csak 16 Boole függvény létezik 2 változóval, megfelelően a 16 lehetséges 4-bites eredménysornak. Ezzel ellentétben a rendes algebrának végtelen számú függvénye van két változóval, amelyek közül egyiket sem lehet leírni egy táblázattal, ami tartalmazza a kimeneteket az összes lehetséges bemenetre, mert minden változó végtelen számú értékeket vehet fel.

A 3-3 (a) ábra egy háromváltozós Boole függvény: M = *f*(A,B,C) igazságtáblázatát mutatja. Ez a függvény egy többségi logikai függvény, ami azt jelenti, hogy ha a bemenetek között a 0 van többségben, akkor az értéke 0, ha az 1, akkor a függvény értéke 1. Bár bármely Boole függvény teljesen meghatározható az igazságtáblázatával, de ahogy a változók száma emelkedik, a jelölési mód egyre jobban fárasztóbb lesz. Ehelyett egy másik jelölést szoktak használni.

3-3-as ábra (a) A többségi függvény igazságtáblázata három változóval (b) Egy áramkör (a)-ra

Hogy lássuk, hogy ez a másik jelölés hogy történik, meg kell jegyezni, hogy egy Boole függvény megadható úgy, hogy megmondjuk, hogy a bemeneti változók mely értékeire lesz a kimenet 1. A 3-3 (a) ábra függvényében a bemeneti változók négyféle kombinációjára lesz M értéke 1.

***[122-125]

A 3-3(a) ábrán látható függvény, M, értéke n esetben lesz 1. Az egyezmények szerint, egy vonalat húzunk a bemenő változó fölé, ha az negálva szerepel. E vonal hiánya azt jelenti, hogy a változó negálatlan. Továbbá a szorzást vagy a pontot fogjuk használni a Logikai AND függvény jelölésére, és a + fogja jelölni a Logikai OR függvényt. Így, például: $A\bar{B}C$ értéke 1, ha A=1 és B=0 és C=1. $A\bar{B}+B\bar{C}$ értéke szintén 1, mikor (A=1 és B=0) vagy (B=1 és C=0). A 3-3(a) ábra négy oszlopa 1 bites kimeneteket ad meg: $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ és ABC. A függvény, M, igaz (azaz 1) ha valamelyik összetevője igaz; így leírhatjuk

$$M=AB\overline{C}+ABC+A\overline{B}C+ABC$$

ez egy rövidebb módja az igazságtábla megadásának. Egy n változós függvény leírható 2^n n változós 'szorzat' összegeként. Ez a formula különösen fontos, mint ahogy azt majd hamarosan látni fogjuk, mert ez oda vezet, hogy a függvényt az alapvető kapuk segítségével valósítjuk meg.

Egy fontos dolog különbséget tenni az absztrakt Logikai függvény és az elektronikus áramkör megvalósítása között. A Logikai függvények változókból állnak, mint például A,B és C, és Logikai operátorokból, mint például AND, OR és NOT. Egy Logikai függvény megadható az igazságtáblájával, vagy megadható így is

$$F=A\overline{B}C+AB\overline{C}$$

Egy Logikai függvény megvalósítható egy elektromos áramkör segítségével (gyakran többféleképpen) felhasználva a bemeneti, kimeneti változókra és a kapukra, mint például AND, OR és NOT, vonatkozó jelöléseket. Általában a Logikai operátorokra az AND, OR és NOT jelölési módszert fogjuk használni, és a kapuk megadására pedig ezt a jelölést: AND, OR és NOT, bár ez gyakran félreérthető.

3.1.3 A Logikai függvények megvalósítása

Ahogy az előbb már említettük egy logikai függvény formalizálása 2^n db szorzat összegeként is megvalósítható. Ha példaképpen megnézzük a 3-3 ábrát, akkor láthatjuk, hogy ez a megoldás tökéletes. A 3-3(a) ábrán a bemenetek A, B és C a bal oldal felől érkeznek, és a függvény kimenetei, pedig a jobb oldalra érkeznek. Mivel a bemeneti változók komplemenseire (negáltjaira) van szükségünk, ezért leágaztatjuk, és átvezetjük az 1-sel, 2-sel és 3-sal jelölt invertálókon. Megfigyelve az ábrát a bejövő ágakból hat darab függőleges vonalat húzunk, melyek közül három darab a bemenő változókhoz, és a három másik vonal a komplemensükhöz kapcsolódik. Ezek a vonalak gondoskodnak a megfelelő forrásról az őket követő kapuk számára. Például az 5-ös kapu, a 6-os és a 7-es kapuk mindegyike az A-t használja bemenetként. Egy aktuális áramkörben ezek a kapuk A-hoz vannak kapcsolva, anélkül hogy közbenső függőleges vezetéket használnánk.

Az áramkör négy AND kaput tartalmaz, egyet az összes taghoz rendelünk, mely kiegyenlíti M-et (azaz az igazságtábla minden sorában van egy bit, és ezen bitek oszlopa adja az eredményt). Minden AND kapu egy sort számít ki az igazságtáblól. Végül, az összes tag össze van kötve egy OR kapuval, és így együtt fogják megadni a végső eredményt.

A 3-3(b) ábrán látható áramkör egy olyan egyezmény alapján van szemléltetve, melyet az egész könyvben használni fogunk: mikor két vonal keresztezi egymást,

akkor azok nincsenek összekapcsolva, ha csak nincs egy pont a találkozásuknál. Például, a kapu kimeneténél hat függőleges vonal, három helyen keresztezi egymást, de csak a C-hez kapcsolódnak. De óvatosnak kell lenni, hiszen néhány szerző másfajta szokást használ.

Hogy letisztázzuk egy áramkör megvalósítását Logikai függvényekkel, vegyük a 3-3 ábrát példaképpen:

1.

Leírjuk a függvény igazságtáblázatát

2. A összes bemenő változónak generáljuk a negáltját
3. Minden taghoz egy AND kaput kaput kapcsolunk, aminek az eredményoszlopában 1 van
4. Összekötjük az AND kapukat a megfelelő inputokkal
5. Az összes AND kapu outputját összekötjük egy OR kapuban

Bár megmutattuk, hogy bármely Logikai függvény megvalósítható NOT, AND és OR kapukkal, gyakran egyszerűbb az áramköröket csak egy típusú kapuval megvalósítani. Szerencsére egyszerű átalakítani az előző algoritmus alkotta áramköröket a tiszta NAND vagy NOR formára. Egy ilyen átalakításhoz csak arra van szükségünk, hogy egy egyszerű kaputípus használatával megvalósítsuk a NOT-ot, AND-et és OR-t. A 3-4 ábra felső sora mutatja, hogy lehet ezt a hármat csak NAND kapukkal megvalósítani, az alsó sor azt mutatja, hogy lehet megvalósítani csak NOR kapukkal. (Ez így megvalósítható, de vannak más utak is.)

Az egyik út a Logikai függvény megvalósításához csak NAND vagy NOR kapukkal, azaz hogy először kövessük ezt a fenti eljárást, amit a NOT, AND és OR kapukkal építhetünk fel. Ezután cseréljük a többbemenetű kapukat ekvivalens áramkörökre kétbementű kapukat használva. Pl.: A+B+C+D-t úgy is számíthatjuk, hogy (A+B)+(C+D) , kétbemenetű OR kapukat használva végül a NOT, AND és OR kapuk a 3-4-es ábra áramköreire cserélendők ki. Bár ez az eljárás nem vezet optimális áramkörökhöz a kapuk minimális száma tekintetében, de megmutatja, hogy a megoldás mindig lehetséges. A NAND és a NOR kapuk is teljesek. Bármely Logikai függvény kiszámítható bármelyik használatával. Egyik másik kapu sem képes erre, ami egy másik ok arra, hogy miért részesítik őket előnyben az áramkörök építésekor.

3.1.4 Áramköri ekvivalencia

Az áramkörtervezők gyakran próbálják csökkenteni a kapuk számát a termékeikben, hogy csökkentsék az alkatrészek számát, nyomtatott áramkör területét, áramfogyasztást, stb. Hogy egy áramkör bopnyolultságát csökkentse a tervezőnek találnia kell egy másik áramkört, amely ugyanazt a

3-4-es ábra. A NOT (a), AND (b) és az OR (c) megvalósítása csak NAND és NOR kapukkal.

függvényt számolja, mint az eredeti, de kevesebb kapuval (vagy esetleg egyszerűbb kapukkal pl.: kétbemenetű kapuk négybemenetű kapuk helyett). Az ekvivalens áramkörök keresése a Logikai algebra hasznos eszköze lehet.

Egy példaként, hogy hogyan használhatjuk a logikai algebrát, tekintsük AB+AC áramkörét és igazságtáblázatát, amelyek 3-5(a)-s ábrán láthatók. Bár még nem beszéltünk róla, a rendes algebra sok szabálya áll a logikai algebrára is. Ebben az esetben AB+AC átalakítható A(B+C)-vé a disztributivitást felhasználva. A 3-5(b)-s ábra mutatja A(B+C) áramkörét és igazságtáblázatát. Mivel a két függvény akkor és csak akkor ekvivalens, ha ugyanaz a kimenetük az összes lehetséges bemenetre, így könnyű látni a 3-5-ös ábra igazságtáblázataiból, hogy A(B+C) ekvivalens AB+AC-vel. Az ekvivalencia miatt 3-5(b) ábra áramköre tisztán jobb, mint a 3-5(a) ábráé, mert kevesebb kaput tartalmaz.

Általában az áramkörtervező egy logikai függvénnyel kezdi, s alkalmazza rá a logikai algebra törvényeit, hogy egyszerűbbet, vele ekvivalenset találjon. A végső függvényből felépíthető az áramkör.

Ezen szemlélet használatához szükségünk van néhány azonosságra a Logikai algebrából. A 3-6-os ábra a legfontosabbat mutatja. Érdemes megemlíteni, hogy minden egyes szabálynak két formája van,

3-5 ábra. Két ekvivalens függvény. AB+AC (a), A(B+C) (b)

amelyek ellentétesek egymással. Az AND és az OR, valamint a 0 és az 1 átváltásánál mindkét forma előállítható a másikból. Minden szabály könnyen kipróbálható azáltal, hogy felépítjük az igazságtáblázatát. A DeMorgan szabályt kivéve az abszorpciós szabályok és disztributív szabály és formájára az eredmények meglehetősen intuitívek. A DeMorgan szabály több, mint két változóval megnövelhető,

pl.: $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$

A DeMorgan szabálya egy alternatív jelölést ajánl. A 3-7(a) ábrán az AND forma tagasással van ábrázolva, az input és az output számára egyaránt. Így egy OR kapu felcserélt inputokkal megegyezik a NAND-beliekkel. A 3-7(b) árbától kezdve (DeMorgan szabály kettős formája) világos, hogy a NOR kapu írható úgy, mint az AND kapu, felcserélhető inputokkal. A DeMorgan szabály mindkét formájának tagadásával elérünk a 3-7(c) és (d) ábráig, melyek az AND és a NOR kapuk ekvivalens ábrázolási módját mutatják. Léteznek még analóg szimbólumok a DeMorgan szabály összetett variális formáira.

Ha a 3-7 ábra azonosságait és az analógiákat többemenetű kapukra használjuk, könnyű konvertálni az igazságtáblázat szorzatok összegének ábrázolási módjait a tiszta NAND és NOR formához. Pl.: megfontolva a 3-8(a) ábra XOR funkcióját. Az állandó szorzatok összegének körforgását a 3-8(b) ábra mutatja.

***[126-129]

Fordította: Bánki Judit V. mat.-I. progmat.

[1]Azért, hogy NAND formára jussunk, az ÉS kapu kimenetét a VAGY kapu bemenetével összekötő vonalat egy inverziót jelentő köröcskével kell megrajzolnunk, ahogy a 3-8(c) ábra mutatja. Végül, a 3-7(a)[2] ábra felhasználásával a 3-8(d)[3] ábrát kapjuk. Az A és B változók az A és B váltzókból kaphatók egy NAND vagy NOR kapu felhsználásával, ha a NAND vagy NOR kapu mindkét bemenetére az A-t vagy a B-t kötjük. Megjegyezzük, hogy az invertáló karikák mozgathatók a vonalak mentén, például az első kapu kimenetéről a második kapu bemenetére (3-8(d) ábra).

Az áramkörök ekvivalenciájával kapcsolatban utolsó esetként egy érdekességet mutatunk meg, mégpedig azt, hogy ugyanaz a fizikai áramköri kapu más-más logikai függvényt valósít meg attól függően, hogy milyen konvenciót használunk. A 3-9(a)[4] ábrán egy meghatározott kapu (F) kimeneteit láthatjuk különböző bemeneti kombinációkra. Mind a bemenet, mind a kimenet voltban van megadva. Ha elfogadjuk azt a konvenciót, hogy a 0 volt a logikai nullát jelenti és a 3.3 volt vagy az 5 volt a logikai egy, melyet pozitív logikának neveznek, akkor a 3-9(b) ábrán látható igazságtáblázatot kapjuk, ami az AND függvény igazságtáblázata. Ha azonban negatív logikát alkalmazunk, amiben a 0 volt a logikai egy és a 3.3 vagy az 5 volt a logikai nulla, akkor a 3-9(c) ábrán látható igazságtáblázatot kapjuk, ami az OR függvény. Látható, hogy a logikai értékek és a feszültség szintek közötti megfeleltetés választása nagyon fontos. A továbbiakban, ha másként nem definiáljuk, pozitív logikát használunk, tehát a "logikai egy", az "igaz" és a "magas" szinonímák, hasonlóan a "logikai nulla", a "hamis" és az "alacsony" szintén ugyanazt jelentik.

3.2 Egyszerű digitális logikai áramkörök

Az előző fejezetben láttuk, hogyan valósíthatók meg az igazságtáblázatok és más egyszerű kapcsolások különálló kapuk felhasználásával. A gyakorlatban csak néhány kapcsolást építenek egyéni kapukból, habár korábban ez volt az elfogadott. Napjainkban, a legtöbb gyártott áramköri lapka egy modul, ami számos elemi kaput tartalmaz. A következő fejezetben közelebbről is megvizsgáljuk ezeket a modulokat és látni fogjuk, hogy hogyan építhetők fel egyéni kapukból és hogyan használhatók.

3.2.1 Integrált áramörök

Kapukat egyenként nem gyártanak és a kereskedelemben sem kaphatók, hanem csak az ezekből felépített modulokat **Integrált Áramköröket**, gyakran **IC**-ket vagy **chip**eket lehet kapni. Az IC egy négyzet alakú pici szilícium lapocska, kb. 5 mm × 5 mm, amelyen számos kapu található. A kis IC-k általában 5-15 mm széles és 20-50 mm hosszú téglalap alakú műanyag vagy kerámia tokozással kaphatók. A hosszabb oldalakkal párhuzamosan kb. 5 mm hosszú lábak vannak, amelyek segítségével az IC-k foglalatba dughatók vagy NYÁK-ra forraszthatók. Minden láb valamelyik kapu kimenetére vagy bemenetére csatlakozik vagy a tápfeszültséghez

[1] 3-6 ábra: A Bool-algebra néhány azonossága
[2] 3-7 ábra: Alternatív szimbólumok néhány kapunál
[3] 3-8 ábra: (a) A XOR igazságtáblája (b)-(d) Ezek az áramkörök valósítják meg
[4] 3-9 ábra: (a) Egy eszköz karakterisztikája. (b) Pozitív logika. (c) Negatív logika.

vagy a földhöz. Azokat a tokokat, amelyeken a lábak két sorban helyezkednek el és belül található az IC, a technikában "**Dual Inline Packages**"-eknek vagy **DIP**-eknek nevezik, de mindenki csak chipeknek hívja őket, elmosva a különbséget a szilícium lapka és a tokozás között. A leggyakrabban használt tokok 14, 16, 18, 20, 22, 24, 28, 40, 64, 68 lábbal rendelkeznek. Nagy chipek esetén a négyzet alapú tokozás a szokásos, a lábak a négyzet mind a négy oldalán, sőt gyakran még a tok alsó részén is megtalálhatók. A chipek osztályokba sorolhatók a bennük lévő kapuk száma alapján, amint ez lejjebb megtalálható. Ez az osztályozási forma nyilvánvalóan nagyon éles, de néha hasznos.
SSI (alacsony mértékű integráció) : 1-10 kapu
MSI (közepes mértékű integráció) : 10 -100 kapu
LSI (nagy mértékű integráció) : 100 - 100000 kapu
VLSI (nagyon nagy mértékű integráció) : 100000 kapunál több
Ezeknek az osztályoknak különböző jellemzőik vannak és a felhasználásaik is különbözők:

Egy SSI chip tipikusan kettő vagy hat független kaput tartalmaz, amelyek mindegyike egyenként használható, az előző fejezetben leírt módon. A 3-10-es ábrán[5] egy közönséges SSI chip sematikus rajza látható, amely négy NAND kaput tartalmaz. Ezen kapuk mindegyike két bemenettel és egy kimenettel rendelekezik, tehát összesen 12 láb szükséges a 4 kapunak. Ezen felül a chip-et el kell látni tápfeszültséggel (V_{cc}) valamint a föld (GND) is szükséges, amelyek el vannak osztva a kapuk számára. A tokozáson a legtöbb esetben egy bemarás található az 1-es láb közelében a lábkiosztás azonosításának megkönnyítése érdekében. A felesleges bonyolultság elkerülése érdekében az áramköri rajzokon a tápfeszültség, a föld és a felesleges kapuk nincsenek feltüntetve. A legtöbb ehhez hasonló chip néhány cent/db áron kapható. Minden SSI chip számos kaput tartalmaz és a lábak száma akár 20 is lehet. A 70-es években a számítógépeket számtalan SSI chip-ből építették, de napjainkban a teljes CPU és a memória bizonyos részeit (cache) egyetlen szilícium lapkára marják rá. A mi vizsgálatainknak megfelelő, ha minden kaput ideálisnak tekintunk, tehát feltesszük, hogy a kimenet azonnal megjelenik, amint a bemenet rendelkezésre áll. A valóságban a chip-eknek véges kapu késleltetésük van, amelybe a jelterjedés sebességét és a kapcsolás idejét is beleértjük. A tipikus kapcsolási idő : 1-10 ns.

A technika jelenlegi szintjén kb. 10 millió tranzisztor helyezhető el egy chipen. Mivel minden áramkör felépíthető NAND kapukból, azt gondolhatnánk, hogy a gyártó teljesen általános célú chipet is gyárthat, ami mondjuk 5 millió NAND kaput tartalmaz. Sajnos egy ilyen chip 15000002 lábat igényelne. Mivel a standard lábtávolság 0.1 inch, ezért egy ilyen chip kb. 18 km hosszú lenne, ami negatívan befolyásolja a piacképességet. Világos, hogy a fejlődés egyetlen lehetséges útja olyan áramkörök készítése, melyeknél a kapu/láb arány magasabb. A következő fejezetben MSI chipeket fogunk tanulmányozni, amelyek több kaput tartalmaznak és segítségükkel a gyakorlatban szükséges bonyolultabb logikai függvények is megvalósíthatók kevesebb külső csatlakozó segítségével.

3.2.2 Kombinációs Hálózatok

Számos digitális elektronikai alkalmazás esetén olyan áramkörre van szükség, amely számos bemenetet és kimenetet tartalmaz, valamint a pillanatnyi bemenetek egyértelműen meghatározzák az aktuális kimenetet. Ezeket az áramköröket

[5] 3-10 ábra: Egy SSI chip 4 kaput tartalmaz

kombinációs hálozatoknak nevezik. Nem minden áramkör rendelekezik ezzel a tulajdonsággal. Például olyan áramkörök, amelyek memória elemeket is tartalmaznak olyan kimenetet is adhatnak, ami a bemeneti változókon kívül a memória tartalmától is függ.

***[130-133]

130-133

Eg igazságtáblát implementáló áramkör, mint az a 3-3(a) ábrán is látható, egy tipikus példája a kombinációs áramköröknek. Ebben a részben megvizsgálunk néhány gyakran használt kombinációs áramkört.

Multiplexerek

A digitális logika szintjén a multiplexer egy áramkör 2^n adatinputtal, egy adatoutputtal és *n* kontrollinputtal, mely kiválasztja az egyik adatinputot. A kiválasztott adatinputot "elkapuzza" (odairányítja) az outputhoz. A 3-11-es ábra[6] egy sematikus diagramja egy 8-inputú multiplexernek. A három kontrollsor, A, B és C, kódol egy 3 bites számot, mely meghatározza, hogy a 8 inputsor melyikét kapuzza el a VAGY kapuba és így az outputhoz. Nem számít, hogy milyen érték van a kontrollsoron, az ÉS kapuk közül 7 mindig 0-t fog kiadni; a másik vagy 0-t vagy 1-et ad a kiválasztott inputsor értékétől függően. Minden ÉS kaput a kontrollinputok egy különböző kombinációja tesz lehetővé. A multiplexer áramkört a 3-11-es ábra mutatja. Amikor áramot és földet adnak hozzá, egy 14 lábú tokba tokozható.

A multiplexert használva a 3-3(a) ábra funkcióinak többségét implementálhatjuk, amint azt a 3-12(b) ábra[7] mutatja. A, B és C minden kombinációjára az adatinputsorok egyikét kiválasztjuk. Mindegyik inputot odavezetik vagy aV_{cc}-be (logikai 1) vagy a földbe (logikai 0). Az inputok odairányításának algoritmusa egyszerű: D_i input ugyanaz, mint az igazságtáblázatban az *i*-edik sorban található érték. A 3-3(a) ábrán a 0., 1., 2. és 4. sorok 0-k, így hozzájuk tartozó inputokat leföldeli; a maradék sorok értéke 1, így ezeket a logikai 1-be vezeti el. Ilyen módon bármilyen 3-változós igazságtáblázat implementálható a 3-12(a) ábra chipjét használva.

Már láttuk, hogyan használhatunk egy multiplexer chipet arra, hogy kiválasszunk számos input közül egyet és hogyan implementál egy igazságtáblát. Ezen kívül alkalmazható még arra is, hogy párhuzamos adatokat konvertáljon sorossá. 8 bit adatot téve az inputsorokra és aztán a kontrollsorokat sorozatosan 000-ról 111-re (bináris) léptetve a 8 bit az outputsorra helyeződik sorozatokban. A párhuzamosból sorossá konvertálás egy tipikus használata pl. egy billentyűzetben van, ahol minden billentyű impliciten definiál egy 7 vagy 8 bites számot, melyet sorosan ki kell vezetni egy telefonvonalon át.

A multiplexer inverze egy **demultiplexer**, mely egyetlen inputjelét a 2^n output egyikéhez vezeti, az *n* kontrollsor értékeitől függően. Ha a kontrollsorokon a bináris érték *k*, a *k* output kerül kiválasztásra.

Dekóderek

Második példaként most egy olyan áramkört nézünk meg, amely egy *n*-bites számot vesz inputként és ezt arra használja, hogy kiválassza (azaz 1-re állítsa) a 2^n outputsorok pontosan egyikét. Egy ilyen áramkört **dekódernek** nevezünk; n=3 esetre

[6] Egy 8-inputú multiplexer áramkör
[7] (a) Egy MSI multiplexer. (b) Ugyanaz a multiplexer átalakítva a funkciók elvégzésére.

a 3-13-as ábrán[8] látható.

Hogy lássuk, hogy egy dekóder mikor lehet hasznos, képzeljünk el egy 8 chipből álló memóriát, melynek mindegyike 1 MB-ot tartalmaz. Az 1-es chip a 0-t az 1 MB-hoz irányítja, a 2-es chip az 1 MB-ot 2 MB-ra, és így tovább. Amikor egy irányítás megjelenik a memóriában, akkor a magas 3 biteket használja fel a 8 chip egyikének kiválasztására. A 3-13-as ábra áramkörét használva ez a 3 bit a 3 input, A, B és C. Az inputoktól függően a D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7 outputsorok pontosan egyike 1, a többi 0. Minden egyes outputsor a 8 memóriachip egyikét hozza működésbe. Mivel csak egy outputsor van 1-re állítva, csak egy chip kerül működésbe.

A 3-13-as ábrán látható áramkör operációja egyszerű. Minden ÉS kapunak 3 inputja van, melyból az első vagy A vagy "A felülvonás"[9], a második vagy B vagy $\underline{B}$, és a harmadik vagy C vagy $\underline{C}$. Minden kaput inputok más kombinációja léptet életbe: D_0-t $\underline{A}$, $\underline{B}$, $\underline{C}$, D_1-et $\underline{A}$, $\underline{B}$, C, és így tovább.

Komparátorok

Egy másik hasznos áramkör a **komparátor**, mely két inputszót hasonlít össze. A 3-14-es ábra egyszerű komparátora két inputot használ, A-t és B-t, mindkettő 4 bit hosszúságú, és 1-et ad, ha ezek egyenlőek és 0-t, ha nem egyenlőek. Az áramkör a KIZÁRÓ VAGY (XOR) kapura épül, mely egy 0-t ad ki, ha az inputok egyenlőek és 1-et, ha nem egyenlőek. Ha a két inputszó egyenlő, mind a négy KIZÁRÓ VAGY kapunak 0-t kell kiadnia. Ez a négy jel aztán összeVAGYolható; ha az eredmény 0, akkor az inputszavak egyenlőek, különben nem. Példánkban egy NEM (NOR) kaput használtuk utolsó lépésként, hogy megfordítsuk a tesztet: az 1 jelenti az egyenlőséget, a 0 a nem egyenlőséget.

Programmed Logic Array

Korábban már láttuk, hogy véletlenszerű funkciók (igazságtáblák) megkonstruálhatók ÉS kapuk számításaival és aztán ezen eredmények összeVAGYolásával. Egy nagyon általános chip az eredmények összesítésének képzésére a **Programmable Logic Array** vagy **PLA**, melynek egy kis példája látható a 3-15-ös ábrán. Ennek a chipnek 12 változós inputsorai vannak. Minden input komplementere belsőleg generálódik, összesen 24 inputot idézve elő. Az áramkör szíve egy 50 ÉS kapuból álló tömb, mely mindegyikének a 24 inputjel bármely részhalmaza lehet potenciálisan inputja. Hogy melyik inputjel melyik ÉS kapuhoz jut el, azt egy, a használó által megadott, 24x50 bites mátrix határozza meg. Az 50 ÉS kapuhoz tartó minden inputsor tartalmaz egy biztosítékot. Amikor a gyárból szállítják, akkor mind az 1200 biztosíték érintetlen. A mátrix beprogramozásához a használó kiégeti a kiválasztott biztosítékokat úgy, hogy a chipekbe magasfeszültséget vezet.

Az áramkör output része hat VAGY kaput tartalmaz, melyek mindegyikének van legfeljebb 50 inputja az ÉS kapuk 50 outputjára válaszképpen. Itt ismét a felhasználó által megadott (50x6-os) mátrix határozza meg, hogy a potenciális kapcsolatok melyike létezzen. A chipnek van 12 inputlába, 6 outputlába, áram és földje, azaz összesen 20 db.

Példaképpen, hogy a PLA-t hogyan lehet használni, vizsgáljuk meg újra a 3-

[8] Egy 3-ból 8-as dekóder áramkör

[9] továbbiakban $\underline{A}$

3(b) ábrájának áramkörét. ennek van 3 inputja, négy ÉS kapuja, egy VAGY kapuja és három invertálója. Ha már megvannak a megfelelő belső kapcsolatok, PLA ugyanezt a funkciót látja el 12 inputjából hármat, 50 ÉS kapujából négyet, 6 VAGY kapujából egyet használva. (A négy kapunak ABC-t, ABC-t, ABC-t és ABC-t kell kiszámítania; a VAGY kapu ezt a négy eredmény fogja fel inputként.) Valójában ugyanezt a PLA-t átalakíthatnánk, hogy egy hasonló összetettségű négy funkció összegét szimultán mődon számitsa ki. Ezeknél az egyszerű funkcióknál az inputváltozók száma a korlátozó tényező; bonyolultabbaknál az ÉS vagy VAGY kapuk.

Bár a fent leírt mezőprogramozható PLA-kat még mindig használják, sok alkalmazásnál inkább a vásárló által készített PLA-kat részesítik előnyben. Ezeket a (nagybani) vásárló tervezi és a gyár állítja elő a vásárló kérésére. Ezek a PLA-k olcsóbbak, mint a mezőprogramozhatók.

Most összehasonlíthatjuk a három különböző módot, melyekrő eddig tárgyaltunk a 3-3(a) igazságtáblázat implementálásáról. SSI összetevőket használva négy chipre van szükségünk. Alternatív módon elég egy MSI multiplexer chip, amint az a 3-12(b) ábrán látható. Végül egy PLA chip negyedét is használhatunk.

***[134-137]

134-Kertész Attila

A DIGITÁLIS LOGIKA SZINTJE

KIZÁRÓ VAGY kapu

3-14. ábra Egy egyszerű 4-bites összehasonlító (comparator).

Az egyszerű áramköröknél az olcsóbb SSI és MSI lapkák előnyösebbek lehetnek.

3.2.3 Aritmetikai áramkörök

Ideje, hogy a fentebb tárgyalt általános-célú MSI áramkörökről tovább menjünk a kombinált MSI áramkörökre, melyeket aritmetikai műveleteknél használunk. Az egyszerű 8-bites léptetővel (shifter) kezdünk, majd megnézzük, hogy milyen az összeadók (adder) felépítése, és végül az aritmetikai logikai egységeket vizsgáljuk, amik központi szerepet játszanak minden számítógépben.

Léptetők (shifters)

Az első aritmetikai MSI áramkörünk egy nyolc-bemenetű és nyolc-kimenetű léptető (lásd 3-16.ábra). A bemenet 8 bitje a D_0,...,D_7-es vonalakon helyezkedik el. A kimenet, aminek épp a bemenet adja át 1 bitjét, az S_0,…, S_7-es vonalakon van. Az irányító vonal, a C, határozza meg a léptetés irányát: 0-át balra és 1-et jobbra.

Hogy lássuk, hogy hogyan működik az áramkör, figyeljük meg az ÉS kapuk (AND gates) párokat az összes bitre, kivéve a végső kapukat. Ha C = 1-el, a párok közül a jobb oldali bekapcsolódik, továbbítva a megfelelő bemeneti bitet a kimenethez. Amiért a jobboldali ÉS kapu a tőle jobbra eső VAGY kapu (OR gate) bemenetéhez van kötve, jobb léptetést hajt végre. Ha C = 0, az ÉS kapuk bal oldala lép működésbe, és hajtja végre a bal léptetést.

Ha ez a biztosíték kiég, B nem bemenet az 1-es ÉS kapuba.

12 x 2 = 24
bemeneti jeladás.

24 bemeneti vonal.

Ha ez a biztosíték kiég, az 1-es ÉS kapu
6 kimenet
nem bemenet, az 5-ös VAGY kapuba.

50 bemeneti vonal

3-15. ábra. 12-bemenetes, 6-kimenetes programozott logikai tömb. A kis négyzetek biztosítékokat jelölnek, melyek kiéghetnek, hogy meghatározzák a kiszámítandó feladatot. A biztosítékok két mátrixot határoznak meg: a felsőt az ÉS kapuk számára, az alsót VAGY kapuknak.

Összeadók (adders)

Egy számítógép, mely nem tud egész számokat összeadni, majdnem elképzelhetetlen. Következésképpen egy összeadást végrehajtó hardver áramkör minden CPU nélkülözhetetlen része. Az 1-bites egészek összeadásának igazság tábláját a 3-17 (a). ábra szemlélteti. Két kimenet létezik: a bemenetek összege, A és B, és a következő helyre (bal felé) átvivő. A számítást végző áramkör összeadó bitjét és átvivő bitjét is a 3-17 (b) ábrán szemléltetjük. Ezt az egyszerű áramkört általában fél-összeadónak (**half adder**) hívják.

3-16. ábra. 1-bites bal-/jobboldali léptető.

KIZÁRÓ VAGY kapu

összeg

átvitel

3-17. ábra. (a) Az 1-bites összeadás igazságtáblája. (b) Egy fél-összeadó áramköre.

Bár a fél-összeadó megfelel két több-bites bemeneti kódszó jobb oldali bitjeinek összeadására, nem fogja elvégezni azokra a bitekre, melyek a kódszó közepén állnak, mert nem kezeli a jobb oldalról történő átvitelt. Helyette a 3-18. Ábrán szereplő teljes összeadóra (full adder) van szükségünk a művelet elvégzéséhez. Miután megszemléltük az áramkört, észrevehetjük, hogy a teljes összeadó két fél-összeadóból áll. Az összeadás kimeneti vonala 1-est ad, ha az A, B és a bevitel páratlan száma 1. A kivitel 1, ha A és B is 1 (bal bemenet a VAGY kapuba) vagy pontosan az egyikük ad 1-et és a beviteli bit szintén 1. A két fél-összeadó együtt állítja elő a bitek összeadását és átvitelét.

bevitel

összeg

kivitel

3-18. ábra (a) A teljes összeadó igazságtáblája. (b) Egy teljes összeadó áramköre.

Ha mondjuk, két 16-bites kódszóra kellene összeadót készíteni, csak meg kellene ismételni a 3-18 (b) ábrán szereplő áramkört 16-szor. A kiviteli bitet a bal oldali szomszédja beviteli bitjeként használnánk. Jobbról a legszélső bit bevitele a 0-hoz van kötve. Ezt a féle összeadót átvitel továbbterjesztő összeadónak (ripple carry adder) nevezzük, mert a legrosszabb esetben, ha 1-et adunk az 111…111 –hez (kettes számrendszerben) az összeadás nem fejeződik be addig, amíg az átvitel el nem jut a jobb szélsőtől a bal szélső bitig. Azok az összeadók, melyek nem rendelkeznek ilyen várakoztatással, és ennél fogva gyorsabbak, szintén léteznek és általában előnyösebbek.

Egy egyszerű példaként a gyorsabb összeadóra, bontsunk szét egy 32-bites összeadót egy alsó és egy felső 16-bites félre. Amikor elkezdődik az összeadás, a felső összeadó még nem tudja elkezdeni munkáját, mert nem tudja a bevitelt a 16-szori összeadáshoz.

Azonban fontoljuk meg ezt a módosítást. A sima felső fél helyett adjunk az összeadónak két felső felet párhuzamosan kapcsolva úgy, hogy megduplázzuk a felső fél hardverét. Így az áramkör három 16-bites összeadóból áll: egy alsó félből és két felsőből, U0 és U1, melyek párhuzamosan futnak. A 0 az U0-ba töltődik be mint átvitel; az 1 az U1-be ugyancsak átvitelként. Így mindkettejük egy időben indul az alsó féllel, de csak az egyikük ad helyes eredményt. 16-szori összeadás után tudni fogjuk, hogy mi a bevitel a felső félbe, így ki tudjuk választani a helyes felső félt a két létező válaszból. Ez a trükk lecsökkenti az összeadási időt két tényezővel. Az ilyen összeadót átvitel kiválasztó összeadónak (carry select adder) hívjuk. Ez a trükk alkalmazható arra, hogy mindkét 16-bites összeadóból 8-biteseket hozzunk létre, és

így tovább.

***[138-141]

Aritmetikai és Logikai egység

A legtöbb számítógépben van az És és Vagy művelet végrehajtására illetve két gépi jel összeadására szolgáló egyszerű áramkör. Az ilyen áramkörök jellegzetessége, hogy n-bitnyi gépi jel kezelésére n db azonos áramkör szolgál a különálló bitek kezelé-sére. A 3-19-es ábra egy egyszerű példa az ilyen áramkörökre amelyeket **Aritmetikai és Logikai Egység-**nek röviden ALU-nak hívnak. Négy művelet bármelyikét ki tudja számolni – A és B, A vagy B, negáció B, A+B – attól függően, hogy a műveletválasztási input vonal két változója melyik művelet kódját tartalmazza binárisan. Így lehet 00, 01, 10, 11. Az itt említett A+B, A és B számtani összegét jelenti, nem a logikai Ést.

Az ábrán lévő ALU bal alsó részén található a 2-bites decoder, amely a két vezérjelből állítja elő a négy művelet kódját. Az F_0 és F_1 értékektől függően a decodolás végén a négy vonalon futó négy művelet közül csak egy lesz engedélyezve.

E vonal beállításai lehetővé teszik, hogy a kijelölt művelet outputja egyenesen a végső VAGY kapuig menjen.

A bal felső sarokban lévő logikai egység végzi az A és B, A vagy B, és negáció B műveleteket, de a decoderből kifutó vonalaktól (enable lines) függően ezen eredmények közül legfeljebb egy juthat a végső Vagy kapuhoz. Mivel a decoder outputjai közül csak egy értéke lesz 1, ezért a négy And kapu közül csak ettől az egytől függ a VAGY kapu értéke. A másik három output értéke 0 lesz, függetlenül az A és B-től.

Ahhoz, hogy A-t és B-t logikai és számtani műveletek inputjaiként használhassuk, lehetőségünk van negáció ENA vagy negáció ENB segítségével valamelyiket a kettő közül 0-vá változtatni. Lehetőség van arra is, hogy megkapjuk A-t INVA beiktatásával.
(INVA, ENA, ENB használatáról a 4. Fejezetben beszélünk) Általában ENA és ENB értéke 1 INVA pedig 0. Ebben az esetben A és B értéke változás nélkül kerül a logikai egységbe.

A jobb alsó sarok feladata: számolja A és B összegét és kezeli a be- és kimenő információt, mivel számos áramkörben ezek valószínűleg össze lesznek vonva, hogy végrehajthassák a teljes szavas műveleteket. A 3-19. ábrán látható áramkört ”bit szeletek” –nek (bit slices) hívják. Ez lehetővé teszi, hogy a számítógéptervezők akármilyen szélességű ALU-t építhessenek. A 3-20. ábra a 8-bit felépítésű ALU-t mutatja 8 1-bites ALU részből összerakva. Az INC jel csak külön hozzáadott művelet esetén hasznos. Működése: Az eredményhez pl.: ad 1-et. Ezzel kiszámíthatóak olyan összegek, mint A+1 vagy A+B+1.

Órajel

Sok áramkörben az utasítások rendje critikusan eltér. Néhol egy utasításnak meg kell előznie egy másikat, néhol két utasítás egyszerre történik. Ezért, hogy a tervezők végrehajthassák a szükséges ütemezést, sok digitális áramkörben használnak órajelet. Az órajel itt egy áramkör, amely olyan impulzusokat sorát bocsájtja ki, amik pontos időközönként követik egymást. Két egymást követő impulzus megfelelő végei közti időtartamotaz órajel ciklusidejének hívjuk. Az impulzus frekvenciája az órajelciklusok 1000 ns – 2 ns –ig terjedő időtartamától függően általában 1 – 500 MHz között van.

A nagyobb pontosság elérése miatt az órajel frekvenciáját rendszerint kristályoszcillátor-ral vezérlik.

A számítógépben több utasítás is végrehajtásra kerülhet egyetlen órajel ciklus alatt. Ha ezeket az utasításokat megfelelő sorrendben kell végrehajtani, akkor az órajelciklust alciklusokra kell felbontani. Az alciklusokban nem az alapórajel fog ütemezni, hanem kitalálták erre egy jobb megoldást. A főórajelvonalat elágaztatják és a mellékágba építenek egy késleltetőáramkört, amely egy másoglagos órajelet generál. Ez az órajel ugyanolyan, mint a főórajel csak a fázisait eltolta a késleltetőáramkör. (3-21/a) ábra.
A 3-21/b ábra az ütemezés diagram, amely négy időpillanatot állít elő négy különálló utasításnak. Az időpillanatok: C1 emelkedése

C1 esése

C2 emelkedése
C2 esése

A különböző utasítások különböző véghez való kötésével végrehajtható lesz a szükséges ütemezésbeli egyeztetés. Ha négynél több órapillanatra van szükség egy órajelcikluson belül, több másodlagos vonalat kell elágaztatni a fővonalból különböző mértékű késleltetőkkel.

Néhány áramkörben az időintervallumokat szívesebben alkalmazzák, mint a különálló időpillanatokat. Pl.: egy utasítás nemcsak C1 emelkedésekor kerülhet végrehajtásra, hanem bármikor amikor C1 fönn van. Egy másik utasítás pedig akkor kerül végrehajtás-ra ha C2 van fönn.

Ha két intevallumnál többre van szükség, akkor több órajelvonalat kell létrehozni vagy a két órajel magas tartományát átfedhetjük egymáson. Így a későbbiekben négy interval-lumot különböztetünk: negáció C1 és negáció C2, negáció C1 és C2, C1 és negáció C2, C1 és C2. Ebben az esetben az órajelek szimmetrikusak, ugyanannyi időt tölt a magas tartományban mint az alacsonyban. (3-21/b)
Aszimmetrikus impulzus előállítása az alapórajelet összeadjuk (ÉS művelettel) abelőle származó késleltetett órajellel. (3-21/c)

Memória

A számítógépek nélkülözhetetlen része a memória. Eddigi ismereteink szerint nem létezhet számítógép memória nélkül. A számítógép a memóriában tárolja az adatokat és a végrahajtásra kerülő utasításokat. A következő részben a memóriarendszer alapvető részeit fogjuk vizsgálni a kapuktól elindulva, hogy lássuk miként működnek és hogy lehet őket kombinálni, hogy terjedelmesebb memóriákat állítsunk elő.

Egybites memória létrehozásához egy áramkörre ami ”emlékszik” az előző input értékre. Az ilyen áramköröket két NOR kapuból állítjuk elő. (3-22/a) Analóg áramköröket NAND kapukból is felépíthetünk. Ezekről nem beszélünk többet mivel teljesen megegyeznek a NOR kapuval.

A 3-22/a ábrán látható áramkört “SR tároló”-nak (latch) hívjuk. Két bemenete van: S: értéket ad a tárolónak (setting) R: törli a tárolót (resetting vagy cleaning). Két kimenete van Q és negáció Q amelyek – később látni fogjuk – kiegészítők. Eltérően a konbinált áramköröktől a tároló kimenetei nem függnek a bemenetektől.

***[142-145]
Nem érkezett meg. Vántus András:h938820

***[146-149]

A flip-flopok maguk 3-27(d)-tipusúak, de a flip-flopban levő inverziós buborékokat a 11-es tűhöz kötött inverter eltörli, igy a flip-flopok emelkedő átmenetbe töltődnek. Mind a 8 törlő szignál sorosan kapcsolt, igy amikor az egyes tű 0 értéket kap, az összes flip-flop 0-ás állapotba kerül. Ha csodálkoznál mért van a 11-es tű invertálva a bemenetnél és minden egyes CK szignálnál újra, lehet hogy egy bemeneti jel nem kap elég áramot, hogy mind a 8 flip-flopot vezérelje; a bemeneti invereter valójában erősítőként funkcionál.

Míg a 3-28(b) ábrán az óra és a tisztító vonalak sorba kötésének az egyik oka a tűk számának csökkentése, ebben a konfigurációban a chip másképp van használva 8 össze nem függő flip-flop-ról (v.-ból).

Egy egyszerű 8 bites regiszterként funkcionál. Így 2 ilyen chip párhuzamosan használva egy 16 bites regisztert alkotnak úgy, hogy az 1-es és a 11-es tűket össze kell kapcsolni.

A 4-es fejezetben részletesen foglalkozunk a regiszterekkel és használatukkal.

3.3.4 *Memóriaszervezés*

Bár mostanra a 3-24-es ábra szimpla 1 bites memóriájától eljutottunk a 3-28(b) ábra 8 bites memóriájáig, nagy memóriák készítéséhez egy másfajta szerkezet szükségeltetik, egy olyan amelyben különálló szavakat meg lehet cimezni. A 3-29-es ábra egy olyan széles körben használt szerkezetet mutat be, ami megfelel ennek a kritériumnak. Ez a példa egy olyan példát illusztrál, aminek 4db 3 bites szava van.

Minden művelet egy teljes 3 bites szót ír vagy olvas. Mig a teljes 12 bites kapacitás alig több, mint a 8-as flip -flopunknál, kevesebb tűt igényel és még fontosabb, hogy könnyű kibőviteni ezt a dizájnt nagy memóriákká.

Bár első látásra a 3-29-es ábra memóriája komplikáltnak tűnhet, valójában egész egyszerű a hagyományos struktúrájának köszönhetően. 8 bemeneti és 3 kimeneti sora van.3 bemenő adat:I_0,I_1 és I_2; 2 a címzéshez: A_0 és A_1;és 3 az ellenőrzéshez: CS→Chip Select, RD tesz különbséget írás és olvasás között és OE→Output Enable.A 3 kimenet az adatoknak van: D_0, D_1 és D_2. Elméletileg ez a memória 14 tűvel megvalósítható, beleértve az áramellátást és a földelést , szemben a 8-as flip-flop 20-as tűigényével.

Ennek a memória chipnek a kiválogatásához ,a külső logikának a CS-t magasra, az RD-t olvasáshoz magasra (logikai 1) és íráshoz alacsonyra (logikai 0) kell állítani. A 2 címsort úgy kell beállitani, hogy megmutassák a 4db 3-bites szó közül melyiket kell olvasni vagy irni. Olvasási műveletnél az adatbemeneti sorok nincsenek használva, de a kiválasztott szó az adatkimeneti sorokra van helyezve. Írás műveletnél az adatbemeneti sorokon levő bitek betöltődnek a kiválasztott memória szóba; az adatkimeneti sorok nincsenek használva.

Most nézzük meg közelebbről, hogy működik a 3-29-es ábra. A memória baloldalán levő 4 szókiválasztó ÉS kapu egy dekódert alkot. Az input inverterek úgy vannak elhelyezve, hogy mindegyik kapu különböző címen érhető el(output magas állásban).----------------------------------
Mindegyik kapu vezérel egy szókiválasztó sort, tetejétől az aljáig, a 0, 1, 2 és 3-as szavakhoz. Amikor a chip ki lett választva iásra, a CS* $\overline{R}$D jelölésű függőleges vonal magas lesz, elérve a 4 írókapu közül egyet, attól függően hogy melyik szókiválasztó sor magas. Az írókapu kimenete vezérli az összes CK jelet a kiválasztott szóhoz, betöltve a bemenő adatokat a flip-flopokba a szóhoz .Az írás csak akkor valósul meg, ha a CS magas az RD alacsony és még akkor is csak az A_0 és A_1 által kiválasztott szó lesz írva, a többi szó nem változik.

Az olvasás hasonlit az íráshoz. A cim megfejtés pontosan ugyanaz mint az írásnál. De most a CS* $\overline{R}$D sor alacsony, igy az összes írókapu munkaképtelen és egyik flip-flop sem módosult. Ehelyett, a kiválasztott szókiválasztó sor eléri a kiválasztott szó Q-bitjeihez csatolt ÉS kapukat. Így a kiválasztott szó outputolja az adatait a kép alján lévő 4 bemenetes OR kapukba, mig a másik 3 szó outputol 0-kat. Következésképpen, az OR kapuk kimenete azonos a kiválasztott szóban tárolt értékkel. A 3 nem kiválasztott szó nem működik közre a kimenetnél.

Bár tudtunk konstruálni egy áramkört, amelyben a 3 OR kapu éppen betáplálódott a 3 kimenőadat sorba, ez a folyamat néha okozott problémát. Különösen, megmutattuk adatbevivő sorok és az adatkivivő sorok különbségét, de az aktuális memóriákban ugyanazok a sorok használódnak. Ha az OR kapukat hozzákötöttük volna az adatkivivő sorokhoz, a chip megpróbálná kivinni az adatot, azaz mindegyik sorhoz egy adott értéket kényszerítene így interferálva a bemenő adattal. Ez okból kívánatos, hogy legyen módunk az OR kapukat az adat kivivő sorokhoz csatlakoztatni olvasáskor, de teljesen szétkapcsolni őket íráskor. Amire szükségünk van, egy elektronikus kapcsoló, ami létre tudja hozni vagy megszakítani a kapcsolatot néhány nanoszekundum alatt.

Szerencsére ilyen kapcsolók léteznek. A 3-30(a) ábra mutatja a jelképet, amit úgy hívnak neminvertáló puffer. Ennek van adatbemenete, adatkimenete és ellenőrző bemenete. Amikor a control bemenet magas, a buffer működése olyan, mint egy vezeték, ahogyan azt mutatja a 3-30(b) ábra is. Amikor a control bemenet alacsony, a buffer működése olyan mint egy nyitott áramkör, ahogy azt a 3-30(c) ábra mutatja; olyan mintha vki elválasztotta volna az adatkimenetet egy drótvágóval az áramkör többi részétől. De ellentétben a drótvágós hasonlattal , a kapcsolatot utólag helyre lehet állítani néhány nanoszekundum alatt a controljel ismételt magasra állításával.

3-30(d) ábra bemutat egy invertáló buffert, amelyik úgy viselkedik mint egy normál inverter, amikor a control magas, és kikapcsolja a kimenetet az áramkörből, amikor a control alacsony. Mindkét fajtája a buffereknek 3 állású szerkezet, mert tud outputolni 0-át,1-et vagy egyiket sem a fentiek közül (nyitott áramkör). A bufferek erősítik is a jeleket, így sok bemenő adatot tudnak egyszerre vezérelni.

Néha használják őket áramkörökben emiatt, még ha a kapcsoló tulajdonságaikra nincs is szükség.

Visszatérve a memória áramkörökhöz, mostanra tisztának kéne lennie, hogy mire való az adatkimeneti soron levő három nem-invertáló buffer. Amikor CS, RD és OE mind magas, a kimeneti feljogosító jel szintén magas, felhatalmazva a buffert és kirakva egy szót a kimeneti vonalakra.

3-27-es ábra.----D zárak és flip-flopok.

3-28-as ábra.(a)----Páros D flip-flop.

(b)----Nyolcas flip-flop.

3-29-es ábra.----Logikai diagram a 4*3-as memóriához. Mindegyik sor egy a 4db 3 bites szavak közül.

Az olvasás vagy írás művelet mindig egy teljes szót olvas vagy ír.

***[150-153]

Ciceri Klári 150-153
H938360

,a kimeneti feljogosító jel szintén magas, felhatalmazva a buffert és kirakni egy szót a kimeneti vonalra. Amikor egy a CS, RD vagy OE közül alacsony, akkor az adatkimenetek kikapcsolódnak az áramkör maradék részéből.

3.3.5.Memória chipek

Jó dolog a 3-29. ábrán látható memóriával kapcsolatban, hogy könnyen lehet bővíteni nagyobb méretekre. Ahogyan lerajzoltuk, a memória 4 X 3-as, azaz 4 szóból áll , szavanként mindegyik 3 bit. Ahhoz,hogy kibővitsük 4 X 8-ra, 5 oszlopnyi, oszloponként 4 billenőkört és 5 bemeneti és kimeneti vonalat kell hozzáadni.Ahhoz hogy a 4 X 3-tól eljussunk a 8 X 3-ig, 4 sort kell hozzáadnunk, soronként 3 billenőkörrel, és természetesen egy A2 címvonalat. Ezzel a fajta felépítéssel a szavak száma a memóriában kettő kell hogy legyen a maximális kihasználtság érdekében, de a bitek száma bármennyi lehet.

A memóriachipek ideális alkalmazási területe az integrált áramkör technológiának, mert ez jól illeszkedik azokhoz a chipekhez, melyeknek a belső felépitése egy ismétlődő 2 dimenziós modell. Amint a technológia fejlődik,a bitek száma amit egy memóriachipre lehet tenni folyamatosan növekszik, tipikusan 18 hónaponta a duplájára (Moore törvénye). A nagyobb chipek nem mindig szorítják háttérbe az elavult kisebbeket, köszönhetően a különböző kapacitás-, sebesség-, erő-,ár- és csatlakozási kényelembeli értékcsökkenésnek. Általában a legnagyobb chipek amellyek most kaphatóak névértékükön felül kaphatóak, és bitenkénti áruk sokkal nagyobb mint a kisebbeké.

Bármilyen adott memóriamérethez többféleképpen szervezhetjük meg a chipet. A 3-31-es ábra egy 4Mbit-es chip két lehetséges elrendezését mutatja be: 512K X 8 és 4096K X 1. (Előrebocsátva, hogy a chipméreteket általában inkább bitben számolják, mint byte-ban,így mi itt felrúgjuk a konvenciókat) A 3-31(a) ábrán 19 címvonal szükséges, hogy meg lehessen címezni egyikét a 2 ^19 byte-nak, és 8 adatvonal, hogy betöltsük, vagy eltároljuk a kiválasztott byte-ot.

A hangsúly itt a terminológián van. Néhány érintkezőn a magasfeszültség idéz elő egy akciót (cselekvést). Néhányon pedig az alacsony idézi elő. Hogy elkerüljük a zürzavart, mi következetesen inkább az fogjuk mondani hogy, egy jel **állítás** (mint azt mondjuk hogy magas vagy alacsony), ami azt jelenti, hogy előidéz egy akciót (cselekvést). Így egyes érintkezőknél az állítás azt fogja jelenteni, hogy magasfeszültségen van, néhányon pedig azt, hogy alacsonyon. Azok az érintkezők, amik alacsonyra vannak állítva jelneveket kapnak ,amelyek magukban foglalnak egy felső vonalat. Így egy jel, aminek a neve CS, magasra van állítva, de aminek CS', az alacsonyra van állítva. Az állítás ellentéte a **tagadás (negálás)**. Amikor semmi különös nem történik az érintkezők negálva vannak.

Most térjünk vissza a memóriachipünkhöz. Amióta a számítógép alaphelyzetben több memóriachipet használ, kell egy jel ami kiválasztja az éppen szükséges chipet,

és csak az válaszol, a többi nem. A CS' (Chip Selecr=chip választó) jel ezt a célt szolgálja. Arra használjuk, hogy aktiválja a chipet. Ugyanakkor szükség van egy módra, hogy megkülönböztessük az írásokat az olvasásoktól. A WE' jel (Write Enable=írás engedélyezés) arra szolgál, hogy megjelölje, hogy az adatot írni kell és nem olvasni. Végül az OE' (Outpu Enable=kimenet engedélyezve) jel azért van, hogy vezesse a kimeneti jeleket. Amikor negálva van, a chip kimenete le van kötve a körről.

A 3-31(b) ábrán egy másik címzési sémát használtak. Belsőleg, ez a chip úgy van megszervezve, mint egy 2048 X 2048-as mátrix 1 bites cellákkal, ami 4 Mbit-et ad. Ahhoz, hogy megcímezzük a chipet először egy sort kell kiválasztani, tehát betesszük a 11-bites számát a címérintkezőkre. Ezután a RAS' (Row Andress Strobe=sor cím…….) jel jön. Ezután az oszlopszámot raljuk a címérintkezőkbe és jön a CAS' (oszlop cím…..). A chip úgy válaszol, hogy vagy elfogad egy adatbitet, vagy kiad egyet.

A ngy memóriachipek gyakran n X n -es mátrixból vannak felépitve, amiknek a címzése sorokkal és oszlopokkal történik. Ez az elrendezés lecsökkenti az érintkezők számát, de lassabbá teszi a chip címzését, mert két címzési ciklus kell: egy a soroknak, egy pedig az oszlopoknak. Hogy visszanyerjük az így elvesztett sebesség egy részét, néhány memóriachipnek egy sorcím után több oszlopcímet is lehet adni, hogy az egy sorban lévő egymás utáni bitekhez hamarabb hozzáférjen.

Évekkel ezelőtt a legnagyobb memóriachipek általában a 3-31(b)-ábrán látható módon voltak megszervezve. Ahogy a memória szavak 8 bitről 32-re és még nagyobbra nőttek, az 1 bit széles memóriachipek kényelmetlenné kezdtek válni. Ahhoz hogy egy 32 bites memóriát építsünk, 32 db 4096K X 1-es chip szükséges párhuzamosan. Ez a 32 chip minimum 16Mb kapacitással rendelkezik, holott 512 X 8-as chipeket használva csak 4 chip kell párhuzamosan, ezzel 2Mb memóriákat is lehet építeni. Hogy elkerüljék a 32 chipet memóriánként, a legtöbb chipgyártónak vannak 1,4,8 és 16 bit széles chipcsaládjai.

3.3.6 RAM-ok és ROM-ok

Az eddig tanulmányozott memóriák mind írhatók és olvashatók voltak. Az ilyen memóriákat RAM-oknak (Random Acces Memory=Véletlen Hozzáférésű Memória) hívjuk, ami egy rossz elnevezés, mert minden memória véletlen hozzáférésű, de már annyira megszokottá vált ez a kifejezés,hogy most már nem változtatnak rajta. A RAM-nak két fajtája van:a statikus és a dinamikus. A statikus RAM-ok (SRAM) belsőleg hasonló áramköröket használnak, mint a mi alap D-billenőkörünk. Ezeknek a memóriáknak az a tulajdonságuk, hogy a tartalmuk addig tárolódik, ameddig be vannak kapcsolva. A statikus RAM-ok nagyon gyorsak. A tipikus elérési idejük néhány ns (nanosecundum). Emiatt a statikus RAM-ok népszerúek 2. szintű cache memóriaként.

A dinamikus RAM-ok (DRAM), ezzel ellentétben nem használnak billenőköröket. Ehelyett a dinamikus RAM egy cellasor, minden cellában egy tranzisztorral és egy kicsi tárolóval. A tárolókat fel lehet tölteni, vagy ki lehet "sütni", ezzel megengedve a 0 és 1 tárolását. Az elektromos töltés azon tulajdonsága miatt, hogy 'elszáll', a dinamikus RAM-okat néhány miliszekundumonként frissiteni (ujratölteni) kekk, hogy megvédjük a benne tárolt adatokat. Amiért a külsú áramköröknek vigyázniuk kell az az, hogy a frissítésre a dinamikus RAM-ok sokkal komplexebb csatlakozási

felülettel rendelkeznek, mint a satikus RAM-ok, de ezt a hátrányát a nagyobb kapacitásával ellensúlyozza.

Amióta a dinamikus RAM-oknak csak egy tranzisztor és egy tároló kell bitenként (ellenben a legjobb statikus RAM-ok 6 tranzisztorával), a dinamikus RAM-ok súrúsége nagyon nagy. Emiatt a központi memóriák majdnem mindig dinamikus RAM-okból vannak megépítve..Habár ennek a nagy kapacitásnak megvan az ára: a dinamikus RAM-ok lassúak (több 10 ns). Így a statikus RAM-ú cache, és a dinamikus RAM-ú központi memória próbálja ötvözni a két memóriafajta jó tulajdonságait.

Sokféle dinamikus RAM chip létezik..A legrégibb fajta amit még ma is használnak az FPM (Fast Page Mode=Gyors Lapozási Mód) DRAM. Belsőleg bitek mátrixaként van megszervezve, és úgy működik, hogy a hardwaretől kér egy sorcímet, majd végigmegy az oszlopcímeken, ahogyan azt leírtuk a RAS' és CAS' jeleknél a 3.31(b) ábránál.

Az FPM DRAM-ot fokozatosan fölváltotta az EDO (Extended Data Output=Kiterjesztett Adatkimenet)DRAM, ami megengedi a második memóriahivatkozás elkezdését mielőtt még az előző befejeződött volna. Ez az egyszerű adatcsatornázás nem tesz gyorsabbá egy egyedüli memóriahivatkozást, de növeli a memória sávszélességét, ami több szót jelent másodpercenként.

Mind az FPM mind az EDO chipek aszinkronúak, ami azt jelenti, hogy a cím és az adatvonal nem egy órajel alatt megy. Ezzel szemben az SDRAM (Synchronous DRAM) hibridje a statikus és a dinamikus RAM-nak,és egy órajel alatt végzi mindkét múveletet. Gyakran használják nagy cachekben, és a jövőben a központi memória kiemelt technológiája lesz.

A RAM nem az egyetlen memóriachip fajta. Sok alkalmazásban , például játékokban, gépekben, és autókban a program és az adatok egy része meg kell hogy maradjon kikapcsolás után is. Ráadásul ha egyszer beépítik, sem az adatot sem a programot nem lehet megváltoztatni soha többet. Ezek a követelmények vezettek a ROM(Read Only Memory=Csak Olvasható Memória) kifejlesztéséhez, ami nem változtatható, vagy törölhető szándékosan vagy máshogy. Az adat a ROM-ba a gyártás során kerül, lényegében megvilágitva egy fotóérzékeny anyagot egy mintán keresztül, ami tartalmazza a kívánt bitmintát, és aztán ezt bevési a felületbe. Az egyedüli mód,hogy megváltoztassunk egy programot a ROM-ban az, hogy ha kicseréljük az egész chipet.

A ROM-ok sokkal olcsóbbak mint a RAM-ok amikor nagy mennyiségben rendeli őket, hogy fedezzék a maszk előállítási költségeit. Ezzel szemben nem rugalmasak, mert a gyártás után nem lehet őket megváltoztatni, és hetekbe telhet amíg egy megrendelt új chip megérkezik. Hogy a cégeknek könnyebbé tegyék a ROM-alapú termékek fejlesztését, kitalálták a PROM-ot.(Programmable ROM=Programozható ROM) A PROM ugyanolyan mint a ROM, azzal a különbséggel, hogy egyszer írható, megspórólva ezzal a rendelés és a szállítás közötti időt. Sok PROM-nak van egy tömbnyi kicsi biztosíték a belsejében. Egy meghatározott biztosítékot ki lehet égetni úgy, hogy megadjuk a sorát és az oszlopát, aztán magasfeszültséget adunk egyik speciális érintkezőjére.

E vonalban a következő találmány az EPROM (Erasable PROM=Törölhető PROM) volt, amit nem csak írni, de törölni is lehetett. Amikor egy kvartz ablak egy EPROM-ban 15 percig erős ultraviola sugárzásnak van kitéve, minden bit értéke 1 lesz. Hogyha sok változtatás várható tervezett időszak alatt, az EPROM-ok messzemenően gazdaságosabbak mint a PROM-ok, mert újra fel lehet használni őket. Az EPROM-ok általában ugyanolyan szervezettségűek mint a statikus RAM-ok.

Például a 4Mbit-es 27C040 EPROM a 3.31(a) ábrán látható elrendezést alkalmazza, ami tipikus statikus RAM elrendezés.

Az EPROM-nál még fejlettebb az EEPROM , amit impulzusokkal lehet törölni ahelyett, hogy be kelljen tenni egy speciális kamrába, és ultraviola sugárzásnak kelljen kitenni. Ráadásul az EEPROM-ot helyben lehet programozni, míg az EPROM-ot csak egy külön erre kifejlesztett EPROM-programozó készülékben lehet.A dolog másik oldalán az áll, hogy nagyobb EEPROM-ok tipikusan csak 1/64 nagyságúak, mint a hagyományos EPROM-ok, és fele olyan gyorsak.Az EEPROM-ok nem vehetik fel a versenyt a DRAM-okkal, vagy az SRAM-okkal, mert azoknál 10-szer lassabbak, és 100-szor kisebbek, és sokkal többe kerülnek. Csak azokban a szituációkban használják őket amikor a feszültség nélküli tároló tulajdonságuk elengedhetetlen.

A legújabb fajtája az EEPROM-nak az úgynevezett fénzmemória. Nem hasonló az EPROM-hoz, amely ultraviola sugárral megvilágítva törölhető, és az EEprom-hoz, amely

3.30 ábra

(a) Egy nem invertáló buffer.
(b) Az (a) következménye, ha a kontrol magas.
(c) Az (a) következménye, ha a kontrol mély.
(d) Egy invertáló buffer.

rajzon: adat be
adat ki
kontrol (ellenőrzés, irányítás)

3.31 ábra

Két megvalósítási útja a 4Mbit-es memória chipnek.

rajzon: 512K X 8 memória chip (4 Mbit)
4096K X 1 memóri chip (4 Mbit)

***[154-157]

Kádas Antal I.KPM
fordítás 154-157. oldal

Az EEPROM-hoz hasonlóan, a flash memória törölhető anélkül, hogy eltávolítanánk az áramkörből. Különböző gyártók állítanak elő kis nyomtatott áramkör-kártyákat 10 MB flash memóriával, hogy "filmként" használják képek tárolására digitális kamerákban és még sok egyéb célra. Manapság a flash memóriát a lemezek kiváltására is használhatjuk, amely nagy előrelépés lenne, alapul véve a 100-nanosecundumos elérési időt. A fő tervezési probléma jelenleg, hogy kb. 10,000 törlés után elhasználódnak, míg a lemezek évekig tartanak, az újraírások számától függetlenül. A különböző fajta memóriák áttekintése a 3-32. ábrán található.

Típus	Kategória	Törlés	Byte módosítási lehetőség	Áramellátás megszűnésével elveszti az adatokat	Tipikus használat
SRAM	Olvasható/írható	Elektromosan	Igen	Igen	Másodszintű cache
DRAM	Olvasható/írható	Elektromosan	Igen	Igen	Fő memória
ROM	Csak olvasható	Nem lehetséges	Nem	Nem	Nagy méretű felhsználás
PROM	Csak olvasható	Nem lehetséges	Nem	Nem	Kis mértű készülékek
EPROM	Főként olvasható	UV fény	Nem	Nem	Készülék protoípusok
EEPROM	Főként olvasható	Elektromosan	Igen	Nem	Készülék protoípusok
Flash	Olvasható/írható	Elektromosan	Nem	Nem	Film digitális kamerákba

3.32 ábra

3.4 CPU LAPKÁK ÉS BUSZOK

Felvértezve információkkal az SSI, MSI és memória chippekről, elkezdhetjük az egészet összerakni, hogy áttekintsük a komplett rendszert. Ebben a fejezetben, először megnézzük a CPU-k néhány általános tulajdonságát a digitális logika szintjéről nézve, beleértve a **lábak kimeneteit** (azt, hogy melyik jel a különböző lábakon mit jelent) is. Amióta a CPU-k annyira összeforrtak a busz felépítésével amit használnak, mi is közlünk bemutatót a buszok felépítéséről ebben a szekcióban. A következő fejezetekben részletes példákkal szolgálunk a CPU-kra és az általuk használt buszokra.

3.4.1 CPU LAPKÁK

Minden modern CPU-t egyetlen chip tartalmaz. Ez teszi az egymásra hatását a legtöbb rendszerrel jól definiálttá. Minden egyes CPU rendelkezik lábakkal, amelyeken keresztül végzi a teljes kommunikációját a külvilággal. Néhány láb kivezeti a jeleket a CPU-ból; mások jeleket fogadnak a külvilágból; néhány mindkettőre képes. Az összes láb funkciójának megértése közben, megtanulhatjuk, hogyan működik együtt a CPU a memóriával és az I/O eszközökkel a digitális logika szintjén.
--------------------új oldal
A CPU-n található lábakat három típusra oszthatjuk: cím, adat és vezérlés. Ezek a lábak hasonló lábakhoz vannak kötve a memórián és az I/O eszközökön párhuzamos kábelek segítségével, melyet busznak hívunk. Egy utasítás végrehajtásához a CPU először a az utasítás memóriacímét elküldi a címlábra. Ekkor igénybe vesz egy vagy több vezérlő lábat, hogy informálja a memóriát, hogy az olvasni akar (pld.) egy szót. A memória válaszol a kért szónak a CPU adat lábaira való küldésével és küld egy jelet, hogy végzett a feladattal. Amikor a CPU látja ezt a jelet, akkor fogadja a szót és végrehajtja az utasítást.
Az utasítás igényelheti adatok írását, olvasását, ebben az esetben a teljes feladatot megismételjük minden további szóra. Lejjebb rátérünk az írás-olvasás részletes ismertetésére. A legfontosabb megérteni, hogy a CPU jeleket állít elő a lábaira és fogad a lábain, így kommunikál a memóriával és az I/O eszközökkel. Semmilyen más kommunikáció nem lehetséges. A két fő paraméter, amely meghatározza a CPU teljesítményét, az a címlábak száma és az adatlábak száma. Egy chip m címlábbal 2^m memória helyet tud megcímezni. Az m általános értéke 16, 20, 32 és 64. Hasonlóan, egy chip n adatlábbal n bites szavakat tud írni, olvasni egy egyszeri műveletben. Az n általános értéke lehet 8, 16 ,

32, 36 vagy 64. Egy CPU-nak 8 adatlábbal négy művelet idejét veszi igénybe egy 32 bites szó olvasása, míg egy másik 32 adatlábbal ezt a műveletet egy művelet ideje alatt végzi el. Tehát a 32 adatlábbal rendelkező chip sokkal gyorsabb, de természetesen még mindig sokkal drágább.
Az adat és cím lábakon kívül, minden egyes CPU-nak van néhány vezérlőlába is. A vezérlő lábak szabályozzák az adatok folyását és időzítését a CPU felé és a CPU-ból és van néhány más felhasználásuk is. Minden CPU-nak van lába a tápellátáshoz (általában +3,3V vagy +5V), földelés, és egy az órajelnek (egy négyzetes hullám), de a többi láb chipről-chipre igen különböző lehet. Mindamellett a vezérlőlábak nagyjából a következő fő kategóriákba sorolhatóak:

1. Busz vezérlés
2. Megszakítások
3. Busz döntés
4. Társprocesszor jelzés
5. Státusz
6. Egyéb vezérlés

Lejjebb röviden ismertetjük az összes kategóriát. Ha rátérünk a Pentium II, UltraSPARC II és a picoJava II chippekre a későbbiekben, akkor több reóészletet fogunk közölni. Az általános CPU chippek a 3-33 ábrán látható jelcsoportokat használják.
A busz vezérlő lábak többnyire kimenetek a CPU-tól a busz felé (ezért bemenetek a memóriának és az I/O chippeknek), melyek közlik, hogy a CPU írni vagy olvasni akarja a memóriát vagy bármi mást akar csinálni.
-------------új oldal
{3-33 ábra Egy általános CPU lábkimeneteinek logikai vázlata. A nyilak jelzik a bemenő jeleket és a kimenő jeleket. A rövid átlós vonalak jelzik, hogy többszörös lábakat használnak. Egy meghatározott CPU-n egy szám adja meg, hogy hányat.}
{*Az ábra szövegei: Középen:* Tipikus Mikroprocesszor; *A nyilak magyarázatai az óra járásával ellentétes irányban:* Címzés, Adat, Busz vezérlés, Megszakítások, Az órajel szimbóluma, A feszültség 5V, Az elektromos föld jele, Egyéb vezérlés, Státusz, Társprocesszor, Busz döntés}
A megszakítás-lábak bemenetek I/O eszközöktől a CPU-ba. A legtöbb rendszerben a CPU utasíthatja az I/O eszközt, hogy elkezdjenek egy műveletet, majd kikapcsoljon és csináljon valami hasznosat, mialatt a lassú I/O eszköz a munkáját végzi. Ha az I/O eszköz elkészült, az I/O vezérlő chip előállít egy jelet valamelyik lábon, hogy eljuttasson egy megszakítási kérelmet a CPU-hoz és leellenőrzi az I/O eszközt, pld. megnézi, hogy történt-e I/O hiba. Néhány CPU-n van egy kimeneti láb, melyre küld egy jelet, hogy tudomásul vette a megszakítást.
A busz döntés vezérlési lábakra a buszon zajló forgalom szabályzásához van szükség, hogy meggátoljuk, hogy két eszköz egyszerre használja a buszt. Döntési célokból, a CPU eszköznek számít.
Néhány CPU-t úgy terveztek, hogy együtt tudjanak működni társprocesszorokkal, úgy mint lebegőpontos egységekkel, de néha grafikus vagy egyéb chippekkel is. Hogy megkönnyítsék a kommunikációt a CPU és a társprocesszor között, speciális lábakkal látták el, hogy különböző kéréseket tudjon előállítani és átadni.
Ezeken a jeleken kívül, van még számos egyéb láb, mellyel néhány CPU rendelkezik. Néhány ezk közül státusz információkat szolgáltat vagy fogad, mások hasznosak a számítógép újraindításához, és ismét mások azért vannak, hogy biztosítsák a kompatibilitást a régebbi I/O chippekkel.

3.4.2 Számítógép buszok

A **busz** egy közönséges elektronikus út összetett eszközök között. A buszokat a funkcióik alapján lehet kategorizálni. Ezeket lehet használni a CPU-n belül, hogy adatokat szállítson az ALU-tól és az ALU-hoz, vagy a CPU-n kívül, hogy a memóriához vagy különböző I/O eszközökhöz kössük őket. Mindkét típusú busznak megvannak a saját követelményei és tulajdonságai.
-------------------új oldal
Ebben a fejezetben és a következőkben a CPU-t a memóriával és az I/O eszközökkel összekötő buszra koncentrálunk. A következő fejezetben a CPU-ban lévő buszokat fogjuk közelebbről megvizsgálni.
A korai személyi számítógépekben volt egy egyedüli külső busz, vagy **rendszerbusz**. Ez 50-100 párhuzamos az alaplapra mart rézkábelből állt, szabályos távolságokban elhelyezett csatlakozókkal, a memória és az I/O kártyák elhelyezéséhez. A modern személyi számítógépek általában rendelkeznek speciális-célú busszal a CPU és a memória és (legalább) egy másik busszal az I/O eszközökhöz. Egy minimális rendszer, egy memória busszal és egy I/O busszal a 3-34. ábrán látható.
{3-34. Ábra Egy számítógép többcélú buszokkal.}
{*Az ábra feliratai:* CPU chip *a CPU chipen belüli feliratok*: Regiszterek, Buszok, ALU, Chippen lévő busz *további feliratok, szintén balról jobbra: felső sor:* Busz vezérlő, Memória busz, Memória *alsó sor:* I/O busz *legalsó sor:* Lemez, Modem, Printer}

A szakirodalomban a buszokat néha “kövér” nyilakkal jelölik, mint ezen az ábrán. A kövér nyíl és a sima vonal egy átlós vonallal megkülönböztetése és a mellette lévő bitek megszámlálása igen aprólékos munka. Ha az összes bit azonos típusú, mondjuk mind cím bit vagy mind adatbit, akkor a rövid átlós vonal az általánosan használt. Ha a cím, adat és vezérlési vonalak összegabalyodtak, akkor a kövér nyilak az általánosak.
Míg a CPU tervezői szabadok, hogy bármilyen típusú buszt használjanak a chippen belül, addig, hogy lehetővé tegyék egy harmadik társaság által gyártott kártya csatlakoztatását a rendszerbuszhoz, jól definiált szabályok vonatkoznak arra, hogy hogyan működnek a buszok, és hogy hogyan kell minden hozzákapcsolt eszköznek engedelmeskednie. Ezeket a szabályokat hívják **busz protokolloknak**.
Emellett, létezni kell mechanikai és elektromos specifikációknak, hogy más társaságok kártyái illeszkedjenek a kártya tartójába és rendelkeznie kell csatlakozókkal, melyek illeszkednek az alaplapon lévőhöz mind mechanikailag, mind feszültségben, időzítésben stb.
A számítástechnikai világban igen sok busz elterjedt. Egy kevés a jobban ismertek közül, jelenlegiek és “történelmiek”, (példákkal) az Omnibus (PDP-8), Unibus (PDP-11), Multibus (8086), IBM PC busz (PC/XT), ISA busz (PC/AT), EISA busz (80386), Microchanel (PS/2). PCI busz (sok PC), SCSI busz (sok PC, és munkaállomás), Nubus (Macintosh), Universal Serial Bus (modern PC-k), FireWire (fogyasztói elektronika), VME busz (fizikai laborfelszerelések), és

***[158-161]

Nem érkezett meg. Szilágyi Tamás:h938285

***[162-165]

3.-37. ábra (a) Olvasás időzítés egy szinkronizált állomáson. (b) Néhány kritikus idő

A T_{ad} és T_{ds} konstrukciók kombinációja azt jelenti, hogy a legrosszabb esetben a memóriának csak 62.5-11-5=46.5 nmp-e lesz attól kezdve, hogy a cím megjelenik addig, amíg elő nem kell állítani az adatot. Mivel 40 nmp elég még a legrosszabb esetben is, egy 40 nmp-es memória mindig tud válaszolni T3 alatt. Egy 50 nmp-es memóriának viszont, be kellene illesztenie egy mp helyzetvárakozást, és T4 alatt válaszolnia.

Az időzítési változatok messze biztosítják azt, hogy a cím legalább 6 nmp-cel azelőtt össze legyen állítva, hogy az MREQ érvényesülne. Ez az idő akkor válhat fontossá, ha az MREQvezeti a chip-szelekciót a memória chipen, mivel néhány memóriaegységhez szükséges lehet egy chip-szelekciót megelőző cím összeállítási idő. Nyilvánvalóan a rendszertervezőnek így nem kellene kiválasztania egy memória-chipet, ami 10 nmp-es összeállítási időt igényel.

A T_m és T_{rl} konstrukció azt jelenti, hogy az MREQ és RD érvényesülni fog a T1 idő csökkenésétől számított 8 nmp-en belül. Legrosszabb esetben a memória-chipnek csak 25+25-8-5=37 nmp-e lesz az MREQ és RD érvényesülése után, hogy eljuttassa adatait az állomásra. Ez a konstrukció ráadásul 40 nmp intervallum igényű, miután a cím tartós.

A T_{mh} és T_{rh} megmondják, meddig tart MREQ és RD tagadása azután, hogy a cím belép. Végül T_{dh} megmondja, meddig kell a memóriának az adatot az állomáson tartania azután, hogy RD el lett utasítva. Ami a mi példánkat, a CPU-t illeti, a memória el tudja mozdítani az állomásról az adatot, mihelyst az RD tagadva van; néhány jelenlegi CPU-n azonban, az adatot kicsit tovább kellene tartósan őrizni.

Rá szeretnénk mutatni arra, hogy a 3-37. ábra egy nagymértékben leegyszerűsített változata a valódi időbeli konstrukciónak. A valóságban sokkal több kritikus idő van, amelyek mindig részletezettek. Mindazonáltal ez egy jó ízelítőt ad arról, hogyan működik egy szinkronizált állomás.

Egy utolsó kérdés, ami fölvetődik, hogy a jelzőberendezés magasabb vagy alacsonyabb rendű legyen. Az állomástervezőkön múlik, hogy eldöntsék, melyik a megfelelőbb, de a választás lényegében tetszés szerinti. Úgy tekinthető ez, mint egy programozói döntés hardware megfelelője, megmutatja a szabad lemezblokkokat egy bit-térképen.

SZINKRONIZÁLT ÁLLOMÁSOK

Bár a szinkronizált állomások könnyen működnek a diszkrét időintervallumaiknak köszönhetően, mégis van néhány problémájuk, például, hogy minden az állomás-idő többszöröseiben működik. Ha egy CPU és memória képes befejezni egy átvitelt 3.1. ciklusokban, ki kell terjeszteniük azt 4.0.-ra, mert a szakaszos ciklusok tiltottak.

Még rosszabb, ha egyszer egy álomásszakaszt kiválasztunk, és a memóriát és I/O kartotékokat erre építünk, mert nehéz a technológiában a jövőbeli fejlesztések előnyeit kihasználni. Például tegyük fel, hogy néhány évvel azután, hogy a 3-37. ábra rendszere épült, új memóriák állnak rendelkezésre, 40 nmp helyett 20 nmp-es belépési idővel. Ezek megszabadultak a várakozóhelyzettől, felgyorsítva így a gépezetet. Aztán tegyük fel, hogy 10 nmp-es memóriák válnak beszerezhetővé. Nem

lenne újabb teljesítmény-növekedés, mert az olvasásra fordított minimális idő 2 kör ebben a tervezetben.

Kicsit más feltételek közé helyezve ezt a tényt, ha egy szinkronizált állomás megoldások heterogén gyűjteményével rendelkezik (néhány gyors és néhány lassú), az állomásnak a leglassabhoz kell igazodnia, és a gyorsak nem tudják kihasználni teljes potenciáljukat.

A kevert technológiát úgy kezeljük, mint szinkronizálatlan állomást, mivel nem “mestere” az időnek, mint a 3-38. ábra is mutatja.

Ahelyett, hogy megpróbálna mindent időzíteni, amikor az állomás-irányító elfogadja a címet, MREQ, RD stb-re van szükség, aztán érvényesít egy speciális jelet, hogy hívni fogja MSYN-t (Szinkronizáció Irányító). Amikor a gép meglátja ezt, végrehajtja a munkát, amilyen gyorsan csak tudja. Amikor elkészül, érvényesíti SSYN-t (Gép Szinkronizáció).

Amint az irányító meglátja SSYN elfogadását, érti, hogy az adat hozzáférhető, így bezárja azokat, aztán elutasítja a címsort az MREQ, RD, MSYN-nel együtt. Amikor a gép látja MSYN elutasítását, érti, hogy a ciklus befejeződött, így elutasítja SSYN-t, újra a kiindulási helyzetben vagyunk, az összes jelzés elutasítva, várjuk a következő utasítást.

A szinkronizálatlan állomások időzítési diagramjai (és néhány szinkronizált állomásé is) nyilakat használnak, amik megmutatják az okokat és hatásokat, mint a 3-38. ábrán. MSYN követelése viszont, a címsorok, MREQ, RD, MSYN elutasítását okozza. Végül MSYN elutasítása, SSYN elutasítását idézi elő, és ez az olvasás vége.

A jelzőrendszert, ami összeilleszti ezt az utat, teljes kézfogásnak nevezzük. Az alapvető rész 4 eseményből áll:

1. MSYN érvényesülése (elfogadása)
2. SSYN elfogadása válaszolva MSYN-re
3. MSYN elutasítása válaszolva SSYN-re
4. SSYN elutasítása válaszolva MSYN elutasítására

Tisztázni kell, hogy a teljes kézfogás időbelisége független. Minden eseményt egy korábbi esemény idéz elő, nem pedig egy időpulzus. Ha egy közelebbi irányító lassú, az semmiképp sincs hatással egy későbbi irányítóra, ami sokkal gyorsabb.

Egy szinkronizálatlan állomás előnyét kellene most tisztázni, de a valóság az, hogy a legtöbb állomás szinkronizált. Ennek az az oka, hogy könnyebb építeni egy szinkronizált rendszert. A CPU csak a saját jelrendszerét fogadja el, a memória csak erre reagál. Nincs visszacsatolás (ok és hatás), de ha jól választjuk ki a komponenseket, minden jól fog működni. Ezenfelül sok pénzt kell befektetni a szinkronizált állomásos technológiába.

3.4.5 ÁLLOMÁSVÁLASZTÁS

Eddig hallgatólagosan elfogadtuk, hogy csak egy állomásirányító van, a CPU. A valóságban az I/O chipnek állomás irányítóvá kell válniuk, olvasni és írni a memóriát, és megszakításokat előidézni. A társprocesszorok szintén állomás irányítóvá válhatnak. A kérdés aztán felvetődik: ”Mi történik, ha egyidőben két vagy több megoldás is állomás irányítóvá szeretne válni?” A válasz az, hogy szükség van

néhány döntő mechanizmusra, hogy megelőzzék a káoszt.

A döntő mechanizmusok lehetnek centralizáltak vagy decentralizáltak. Először vizsgáljuk meg a centralizáltat. Ennek egy egyszerű formáját mutatja 3-39(a) ábra. Ezen a vázlaton egy állomás irányító eldönti , mi következzen. Sok CPU-ba ez be van építve, de néha külön chipre van szükség. Az állomás tartalmaz egy elkülönített QR kérelmi vonalat, ami egy vagy több megoldás által kerül elfogadásra. A bírálórendszernek semmiképp nem mondja meg, mennyi megoldást kért az állomás, csak kategóriákat tud megkülönböztetni, amelyek néhánya kérelem és van néhány nem kérelem.

Amikor a mechanizmus látja az állomás kérelmét, kiad egy engedélyt, amivel érvényesíti az állomás engedélyező vonalát. Ez a vonal keresztül húzódik az I/O megoldások sorozatán, úgy, mint a karácsonyfaégők fűzére. Amikor a megoldás fizikailag bezáródott, a mechanizmus az engedélyt ellenőrzi, elkészült-e a kérelem. Ha így van, átviszi azt az állomáson, de közben terjeszti lefelé a vonalon. Ha nincs kérelem, az engedélyt a vonalon a következő állomáshoz terjeszti, amely ugyanezt az utat teszi meg, és így tovább, amíg néhány megoldás elfogadja az engedélyt és átveszi az állomást. Ezen a vázlaton ezt "jó láncolatnak" nevezik. Van egy olyan tulajdonsága, hogy a megoldások elsőbbsége attól függ, hogyan záródnak a mechanizmushoz. A legzártabb megoldás nyer.

Az előbbségek megkerülése a mechanizmustól való távolságon alapul, sok állomásnak többszörös elsőbbségi szintjei vannak. Minden elsőbbségi szintre van egy állomás kérelem és egy engedélyvonal. A 3-39(b) ábra egyikének 2 szintje van 1 és 2 (de valamelyiknek van 4,8 vagy 16 szintje is). Az összes megoldás csatlakozik az állomás kérelmi szintjének egyikéhez, az időkritikus megoldások csatlakoznak az elsőbbrendűekhez. A 3-39(b) megoldás; 1, 2 és 4 használ 1 prioritást, míg a 3 és 5,2 prioritást használnak.

Ha többszörös prioritású szinteket igényeltek ugyanabban az időben, a mechanizmus engedélyt csak a legmagasabb prioritásúnak ad ki. Az ugyanolyan prioritású megoldások között a láncolatot használja. A 3-39(b) ábrán, ellentmondás esetében, a 2. megoldás kiüti a 4-et, ami kiüti a 3-at.

***[166-173]

Az 5-ös egységnek van a legkisebb prioritása mert a legkisebb prioritású láncolat végén helyezkedik el.

Mellékesen, technikai szempontból nem szükséges a 2. szintű engedélyező buszvonal 1-es és 2-es egységeken keresztüli sorbakötése, mert ezek rajta keresztül nem küldhetnek kérő utasítást. Azonban a gyakorlati kivitelezésben egyszerűbb az összes engedélyező vonalat az összes egységbe bekötni, semminthogy az egységek prioritásától függő speciális bekötést alkalmazni.

Néhány csatornakiválasztó rendelkezik egy harmadik vonallal is, amelyre egy egység akkor küld megkérő jelet, mikor elfogadott egy engedélyezést és rákapcsolódott a buszra. Ahogy hatályosította ezen nyugtázó vonalat, a kérő és engedélyező vonalak negálhatóak. Ennek eredményképp, míg az első egység a buszt használja, a többi egység is kérő utasítást küldhet. Mire a folyamatban lévő átvitel befejeződik, már a következő bus-master ki is választódik. Ez rögtön indulhat amint a nyugtázó vonal hatálytalanítódik, amikor is elkezdődhet a következő kiválasztási ciklus. Ez a séma egy extra busz-vonalat és egységenként több logikát követel, de jobban használja a busz-ciklusokat.

Az olyan rendszerekben, ahol a memória a főbuszon van a CPU-nak szinte minden ciklusban versenyeznie kell a buszért az összes I/O egységgel. Ezen helyzet általános megoldása, hogy a CPU kapja a legalacsonyabb prioritást, így csak akkor fér hozzá a buszhoz, mikor azt senki más nem használja. Az alapötlet itt az, hogy a CPU mindig várhat, míg az I/O egységeknek gyakran gyorsan hozzá kell jutniuk a buszhoz különben a beérkező adatok elvésznek. A nagy sebességgel forgó lemezek nem várhatnak. Ezt a problémát számos modern számítógép-rendszerben úgy kerülik el, hogy a memóriát az I/O egységektől különálló buszra kapcsolják, hogy ne kelljen a busz eléréséért versenyezniük.

A decentzralizált buszkiválasztás is megvalósítható. Például, egy számítógépnek lehet 16 prioritás-sorrendbe állított busz-kérő vonala. Mikor egy egység a buszt kívánja használni , akkor hataályba helyezi a kérővonalát. Az összes egység szemmel tartja az összes kérővonalat, így minden busz-ciklus végén minden egység tudja, hogy a legmagasabb prioritással rendelkező kérő volt-e, és hogy a következő ciklus alatt használhatja-e a buszt. A centralizált kiválasztáshoz hasonlítva, ez a kiválasztási módszer több busz-vonalat követel, de kivédi a kiválasztó lehetséges túlterhelését. Ugyanakkor, az egységek számát a kérővonalak számára korlátozza.

A decentralizált busz-kiválasztás egy másik fajtája, amelyet a 3-40.-es ábrán mutatunk, csak 3 vonalat használ, függetlenül a jelenlévő egységek számától. Az első busz-vonal, egy a busz kérését szolgáló becstlakoztatott-VAGY vonal. A második busz-vonalat FOGLALT-nak nevezik, és az éppen alkalmazott bus-master hozza működésbe. A harmadik vonalat a busz kiválasztására használják. Ezt az összes egységen keresztülvezető megszakításprioritás-láncba kötik. A lánc fejét egy 5V-os feszültségforrás segítségével tartják engedélyezett állapotban.

3-40.-es ábra

Mikor egyik egységnek sincs szüksége a buszra, az engedélyezett kiválasztó-vonal az összes egységnek továbbítódik. A busz eléréséhez egy egység először

ellenőrzi, hogy a busz szabad-e, és hogy az általa fogadott kiválasztó jel engedélyezett-e. Ha a BEMENET-ét hatálytalanították, akkor nem válhat bus-masterré, éas így hatálytalanítja a KIMENET-ét. Ha azonban a BEMENET engedélyezett, akkor az egység negálja a KIMENET-ét, amely arra készteti folyásirányú szomszédját, hogy saját BEMENET-ét nem engedélyezetttnek lássa és saját KIMENET-ét is hatálytalanítsa. Ezáltal minden a folyásirányban lévő minden egység lezárt BEMENET-et lát, és ennek megfelelően lezárja kimenetét is. Mikor elül a vihar, csak egy egységenek lesz nyitott BEMENET-e és zárt KIMENETE. Ez az egység lesz a bus-master, engedélyezi a FOGLALT-at és a KIMENET-et, és elkezdi az átvitelt.

Némi gondolkodás után rájöhetünk, hogy a buszt elérni kívánó gységek közül a legbaloldalibb kapja azt meg. Így tehát ez az elképzelés hasonlatos azeredeti megszakításprioritás-láncolatos kiválsztáshoz, eltekintve attól, hogy nem kell hozzá kiválasztó, azaz gyorsabb, olcsóbb, és a kiválaszó hibáitól sem függő.

3.4.6 BUSZ MŰVELETEK

Egészen mostanáig, csak általános busz-ciklusokról beszéltünk. Ciklusokról, amelyek egy olyan mesterrel (általában a CPU) rendelkeztek, amely egy szolgából (általában a memória) olvas, vagy éppenséggel abba ír. Vlójában sok másféle busz-ciklus létezik. Most megtekintünk néhányat ezek közül.

Normális esetben csak egy szó kerül átvitelre. Azonban, mikor cache-elést használunk kívánatos rögtön egy teljes cache-vonalra szert tenni (pl. 16 egymást követő 32-bites szavak). A blokkátvitelek gyakran hatásosabbé tehetők, mint az egymás után következő különálló átvitelek. Mikor elkezdődik egy blokkolvasás, a bus-master közli a szolgával, hogy hány szó átvitelére fog sor kerülni, például úgy, hogy a szavak számát az ADAT vonalakra teszi T1 során. Egyetlen szó visszaküldése helyett a szolga minden egyes ciklus során kihelyez egy szót, addig amíg a szám el nem fogy. A 3-41.-es ábra a 3-37 (a) ábra módosított változatát mutatja, de most egy extra jellel, a BLOCK FELÜLVONÁS-sal, amelyre a blokkátvitel megkérésének jelölése végett van szükség. Ebben a példában 4 szó blokkolvasása 12 ciklus helyett csak 6 ciklus hosszúságú.

3-41. ábra

Másfajta busz-ciklusok is léteznek. Például, egy multiprocesszoros rendszerben, ahol ugyanazon buszra 2 vagy több Cpu is rácsatlakozik, gyakran szükséges anak biztosítása, hogy egyszerre csak egy CPU használjon a memóriában lévő kritikus adatstrukturákat. Ennek tipikus megoldása egy a memóriában lévő változó, amely 0 mikor semelyik CPU sem használja az adat strukturákat, s 1 mikor használatban van. Ha egy CPU hozzá szeretne férni az adatstruktúrához, el kell olvasnia a változót, és ha az 0, 1-re állítania. A baj az, hogy egy kis balszerencsével, két CPU is olvashatja azt konszekutiv buszciklusokban. Ha mindkettő 0-nak látja a változót, akkor mindkettő átáálítja 1-esre és azt gondolja, hogy ő az eyetlen Cpu, amely az adatstruktúrát használja. Ez az eseménysor káoszhoz vezet.

Ezen helyzetet kivédendő, a multiprocesszoros rendszereknek gyakran van egy speciális olvas-módosít-ír busz-cilkusa, amely akármelyik CPU-nak engedélyezi egy szó olvasását a memóriából, ezen szó felülvizsgálatát és módosítását, majd memóriába való visszaírását, s mindezt a busz átengedése nélkül. Az ilyen típusú

ciklus megakadályozza a versengő CPU-kat a busz használatában, és hogy eképp beleavatkozzanak az első CPU működésébe.

A busz-ciklus egy másik fontos fajtája a megszakítások kezelésére jött létre. Mikor a CPU egy I/O egységnek valaminek a végrehajtására ad ki parancsot, általában egy megszakításra számít a munka elvégzése után. A megszakítás jelzéséhez szükség van a buszra.

Mivel összetett egységek egyidejüleg akarhatnak megszakítást előidézni, ugyanazon kiválasztási problémákkal találjuk magunkat itt is szembe, mint az általános busz-ciklusoknál. A szokásos megoldás az, hogy prioritásokat jelölünk ki az egységeknek, és hogy egy centralizált kiválasztót használnak, amely a legidőkritikusabb egységeknek ad prioritást. Standard megszakytásszabályozó chipek léteznek és széles körben használják őket. Az IBM PC és minden utóda az Intel 8259A chipet használja. Ezt a 3-42.-es ábrán láthatjuk.

Akár 8 I/O szabályzó chip csatlakoztatható közvetlenül a 8259A 8 IRX (megszakítás kérés) bemenetére. Mikor akármelyik ezen egységek közül megszakíytást akar előidézni, akkot kinyitja bemeneti vonalát. Ha egy vagy több bemenet kinyilik, a 8259A kinyitja az INT (megszakítás)-et, amely közvetlenül a CPU megszakítás lábát hajtja meg. Ha a CPU képes a megszakítás kezelésére, akkor egy pulzust küld vissza a 8259a-nak az INTA (megszakításnyugtázás)-n. Enél a pontnál a 8259A-nak meg kell határoznia, hogy melyik bemenet okozta a megszakítást aképpen, hogy kihelyezi annak a bemenetnek a számát az adatbuszra. Ehhez a művelethez egy speciális busz-ciklusra van szükség. A CPU hardver ezt a számot hazsnálja a pointer-, vagy megszakításvektor-táblába való beiktatáshoz, és ezáltal a megszakyítás kiszolgálásához szükséges futtatandó eljárás címénak a megtalálásához.

A 8259A-nak számos olyan belső regisztere van, amelyeket a CPU közönséges busz-ciklusok és az RD FELÜLVONÁS (olvasás), W FELÜLVONÁS (ÍRÁS), CS FELÜLVONÁS (chipkiválasztás) és A0 FELÜLVONÁS lábak használatával írni és olvasni tud. Mikor a szoftver már kezelte a megszakítást és kész következő fogadására, egy speciális kódot ír a regiszterek egyxikébe, amely azt idézi elő, hogy a 8259A lezárja az INT-et, hacsak nincs még egy folyamatban lévő megszakítása. Ezen regiszterek úgy is írhatók, hogy a 8259A-t a számos mód egyikébe tegyék, megszakításkészleteket kimaszkoljanak, és más különlegességeket lehetővé tegyenek.

Ha több mint 8 I/O egység van jelen, akor a 8259A-kat kaszkádba kapcsolhatjuk. A legszélsőségesebb esetben mind a 8 bemenet 8 másik 8259A kimenetére csatlakoztatható, lehetővé téve 64 I/O egység használatát egy kétfokozatú megszakításhálózatban. A 8259A-nak van néhány lába ezen kaszkádolás kezelésére szánva, amelyekről az egyszerűség kedvéért nem szólunk.

Jóllehet a busz-tervezés témáját mégcsak megközelítőleg senm merítettükk ki, az előzőekben közreadott anyag elég alapot ad a busz müködési lényegének és a CPU-k és buszok közremüködésének megértéséhez. Most pedig lépjünk tovább az általánostól a specifikus felé és tekintsünk meg tényleges CPU-k és buszaik néhány példáját

3.5 PÉLDA CPU CHIPEK

ebben a részben kis részletességgel, a hardverszinten fogjuk vizsgálni a Pentium II-t, az UltraSPARC II-t, és a picoJava CPU chipeket.

3.5.1 A PENTIUM II.

A PII az eredeti IBM PC-ben használt 8088 CPU közvetlen leszármazottja. Habár a PII 7.5 millió **(ha a cikk amerikai, lehet 7.5 milliárd is)** tranzisztorjával messze álla 29.000 tranzisztoros 8088-tól, mégis teljesen kompatibilis a 8088-al és módosítás nélkül képes futtatnia 8088 bináris programjait (a közbeeső processzorokra írt programokat nem is említve).

Szoftver szempontból a PII teljességgel egy 32 bites gép. Ugyanazzal a használóiszintű ISA-val rendelkezik, mint a 80386, 80486, Pentium, és Pentium Pro, beleértve ugyanazokat a regisztereket, ugyanazokat a parancsokat, és az IEEE 754 lebegőpontos standard teljes on-chip megvalósítását.

Hardveri megközelítésből a PII valamivel több, hiszen 64GB létező memóriát tud megcímezni, és 64 bites egységekben képes adatokat a memóriájából és memóriájaba átvinni. Habár a programozó nem tudja a 64 bites átviteleket szemmel kísérni, mégis ezek gyorsabbá teszik a gépet, mint egy tisztán 32 bites gép.

Belsőleg, mikrotechnológiai szinten, a PII alapjában véve egy Ppro az MMX utasításokkal kibővítve. Az ISA szintű paprancsokat jó előre behívja a memóriából és RISC-szerű mikroműveletekre töri fel. Ezeket a mikroműveleteket egy pufferban tárolja, és amint valamelyikük elegendő erőforrással rendelkezik a kivitelezéshez, az elindulhat. Összetett mikroműveletek ugyanazon cikluson belül indíthatók, így a PII-t egy skálafeletti, szuperskaláris géppé téve.

A PI-nek kétszintű cache-je van. Rendelkezik egy pár, 16KB az utasításoknak és 16KB az adatoknak, on-chip cache-sel, és egy 512 KB-os egyesített másodikszintű cache-sel. A cache-vonal 32 Byte méretű. A másodikszintű cache a CPU órafrekvenciájának a felével működik. A CPU órák 233 MHz-en vagy afelett hozzáférhetőek.

A PII rendszerekben két elsődleges külső, externális buszt használnak. Mindkettő szinkronikus. A memória-buszt a fő DRAM eléréséhez, a PCI buszt az I/O egységekkel való kommunikációra használják. Néha egy örökség((????) legacy) (azaz ősi) buszt is hozzácsatolnak a PCI buszhoz, ezzel a régi preifériaegységek csatlakoztatását is lehetővé téve.

A PII rendszernek lehet 1 vagy 2 CPU-ja, amelyek közös memórián osztozkodnak. A két CPU-s rendszernél fennáll annak a veszélye, hogy ha egy szót beolvas egy cache-be és ott módosul, anélkül, hogy visszíródna a memóriába, és ha a másik CPU megpróbálkozik a szó olvasásával, akkor helytelen értéket fog kapni. Speciális védelemről (snooping) ezen probléma kivédése érdekében.

Egy alapvető különbség a PII és minden elődje között a kiszerelése. Egészen a 8088-tól kezdődően, a PPro-n keresztül és azt is beleértve, minden Intel CPU normál chip volt, lábakkal az oldalán vagy az alján, amelyeket egy foglalatba lehetett csatlakoztatni. Ettől eltérően, a PII az Intel által SEC (Egyélű Cartridge, Single Edge Cartridge)-nek nevezett burkolatban jelenik meg. Ahogy azt a 3-43.-as ábrán láthatjuk, egy SEC az egy igen nagyméretű műanyag doboz, amely tartalmazza a CPU-t, a második szintű cache-t, egy élcsatlakozót a jelek továbbításához. A PII SEC-nek 242 csatlakozója van.

3-43. ábra

Bár az Intelnek kétségtelenül jó oka van arra, hogy ezt a kiszerelési modellt válassza, mégis ez olyan vomatkozásban okozott problémát, amit az Intel nem látott előre. Vitathatatlan, hogy sok vásárló bír azzal a szokással, hogy kicsavarozza gépét csak azért, hogy megkeresse a CPU chipet. Az első leszállított PII-ben a vásárlók nem találták a CPU-t és hangosan reklamáltak (Az én számítógépemnek nincs CPU-ja!). Az Intel úgy oldotta meg ezt a problémát, hogy a CPU chip képét (valójában egy hologramot) ragasztották minden következőleg leszállított SEC-re.

Az energiaellátás fontos kérdés a PII-nél. A leadott hő mennyisége az órafrekvenciától függ, de kb. a 30-50 W-os sávba esik. Ez óriási menyiség egy számítógép chipnek. Hogy valami elképzelése legyen arról, milyen is az az 50 W, tegye a kezét egy már jó ideje működő izzólámpa közelébe (de ne ré). Következésképpen, a SEC-et úgy szerelték ki, hogy befogadjon egy hőelnyelőt, amelynek a létrehozott hő eloszlatása a feladata. Ez a sajátságosság azt jelenti, hogy mikor egy PII leélte CPU-ként való használhatóságának idejét, akkor még mindig hasznosíthatjuk, mint tábori tűzhely.

A fizika törvényeinek megfelelően, akármi ami sok hőt ad le, sok energiát kell hogy felvegyen. Egy hordozható számítógépnél, korlátozott akkumullátor töltettel, a nagy energiafelhasználás nem kívánatos. Ezen kérdést megcélozva, az Intel lehetővé tette a CPU álomba merülését, mikor az nincs használat alatt, és mély álomva merülését, mikor valószínűsíthető, hogy így is marad egy ideig. A mélyálom állapotában a cache és regiszterértékek megőrződnek, de az óra és minden belső egység kikapcsolódik. Arról azonban nem szólnak a beszámolók, hogy a PI mélyálom állapotában álmodik-e.

A PII LOGIKAI LÁBKIOSZTÁSA

A SEC 242 érintős élcsatlakozójának a kiosztása tartalmaz 170 jel és 27 energiacsatlakozást (különböző feszültségszintekkel), 35 föld, és 10 tartalékcsatlakozást, jövőbeli használatra. A logikai jelek némelyike két, vagy több lábat is használ (mint például a megkért memóriacím), így csak 53 különböző van belőlük. Egy valamelyest egyszerűsített logikai lábkiosztást láthatunk a 3-44.-s ábrán. Az ábra baloldalán helyezkedik el a memóriabusz-jelek 6 fő csoportja, a jobb oldalon többféle jel található vegyesen. A baloldali esetben megadott nevek a tényleges Intel jelnevek. A jobboldaliak pedig összetett, kapcsolódó jelek gyűjtőnevei.

Az Intel olyan elnevezési szokást alkalmaz, amit fontos megérteni. Mivel manapság a chiptervezés számítógépek használatával folyik, szükség van a jelnevek ASCII szövegben való megjelenítésének lehetőségére. A felülvonások használata, az alacsonyszintre kapcsolt jelek indikálására, túl bonyolúlt, ezért az Intel ehelyett # karaktert helyezi a név után. Így tehát a BPRI-t BPRI# formában fejezik ki. Ahogy azt az ábrából láthatjuk a legtöbb PII jel alacsony szintű.

Most pedig vizsgáljuk meg magukat a jeleket, a busz-jelekkel kezdve. A jelek első csoportját a busz megkérésére (azaz a busz kiválasztás elvégzésére) használják. A BPRI# lehetővé teszi, hogy egy egység magas prioritású megkérést tegyen, melynek elsőbbsége van egy általánossal szemben. A LOCK# megengedi egy CPU-nak a busz lezárását, hogy ezáltal megakadályozza a másikat a rácsatlakozásban, addig amíg készen nincs.

Amint megszerezte a busz feletti tulajdonjogot, a CPU, vagy más bus-master buszmegkéréseket tehet a jelek következő csoportjának a használatával. A címek 36

bitesek, de az alacsonyrendű (jobbszélső) 3 bitnek mindig nullának kell lennie, ezért nincsenek kijelölt lábaik, tehát az A# csak 33 lába van. Minden átvitel 8 bájtos és 8 bájtos határon rendeződik el. 36 cím bittel a maximálisan megcímezhető memória az 2 36-on, ami 64 gigabájt.

Mikor egy cím a buszra kerül, az ADS# jelet használják, hogy a célponttal (például a memória) közöljék, hogy a címvonalak érvényesek. A busz-ciklus típusa (például egy szó olvasása, vagy egy blokk írása) az REQ# vonalakon továbbítódik. Két paritásjel védi az A#-t és egy védi az ADS#-t és az REQ#-t. Az öt hibavonalat a szolga a paritáshibák, míg a többi egység a más jellegű hibák jelentésére használja.

3-44-es ábra

A Snoop csoportot a multiprocesszoros rendszerekben annak engedélyezésére használják, hogy egy CPU megtudhassa, hogy egy számára szükséges szó a másik CPU cache-jében van-e. Azt, hogy a PII hogyan snoop-ol a nyolcadik fejezetben fogjuk leírni.

A Válasz csoport olyan jeleket foglal magában, amelyeket a szolga használ a mesternek való visszajelentéshez. Az RS# tartalmazza az állapotkódot. Az ATRDY# azt jelöli, hogy a szolga (a célpont) kész a mestertől jövő adat befogadására. Ezek a jelek egyben paritás-ellenőrzöttek is.

Az utolsó busz-csoport a tényleges adatátvitelre való. A D#-et használják a 8 adatbájt buszrahelyezésére. Mikor odakerültek, a DRDY# aktivizálódik, hogy bejelentse jelenlétüket. A DBSY#-et arra használják, hogy közölje a nagyvilággal, hogy a busz jelenleg foglalt.

A RESET#-et a CPU vész esetén való resetelésére használják. A PII úgy is konfigurálható, hogy a megszakításokat a 8088-sal megegyező módon használja (a visszafelé kompatibilitás végett), vagy használhat egy új megszakítási rendszert egy APIC (Advanced Programmable Interrupt Controller, Fejlett Programozható Megszakítás Vezérlő) nevezetű eszköz alkalmazása mellett.

***[174-177]
Nem érkezett meg. Márki-Zay Dániel: h938312

***[178-181]

3-47.ábra. Egy UltraSPARC II rendszer magjának főbb jellemzői

A 256 a leggyakrabban használt utaítássor, és a 256 a leggyakrabban használt adatsor, ezek az elsőszintű cache-ben vannak. A cache line-okat gyakran használják, de amik nem férnek be az elsőszintű cache-be, azok a második szintű cache-ban vannak tárolva. Ez a cache tartalmazza véletlenszerűen összekeverve mind az adatsorokat, mind az utaítássorokat. Mindezek a "Level 2 cache data" címkézésű téglalapban vannak tárolva.
A rendszernek kell nyomon követnie, hogy melyik sor tartozik a második szintű cache-be. Ez az információ a második SRAM-ban van őrizve, "Level 2 cache tags" címen.

Ha nincs az adat az elsőszintű cach-ben, akkor a CPU elküldi az azonósítóját (vagyis a Tag address-t) a második szintű cache-be. Az ismétlés (azaz a Tag data) információt szolgáltat a CPU számára, a CPU pedig közli, hogy a sor vajon a második szintű cache-ben van-e, és ha igen, akkor milyen állapotban. Ha a sor ott van, akkor a CPU megy és megszerzi. Az adatszállító 16 byte széles, így négy ciklus szükséges ahhoz, hogy egy teljes sort az elsőszintű cache-be szállítson a másodikból

Ha a cache line nincs a második szintű cache-ben, akkor muszáj a főmemóriából az UPA interface-be hozni. Az UltraSPARC II UPA egy központosított vezérlővel van kivitelezve. A cím és a vezérlő jelek a CPU-ból mennek a kontrollerbe (ha több CPU van, akkor mindegyikből). Miután a CPU megkapja az engedélyt, kiküldi a címet a memory address lábakon, megadja a kérés típusát és érvényesíti a címet az address valid lábakon. (Ezek a lábak kétirányúak, mivel az UltraSPARC multiprocesszorának többi CPU-inak is szükséges hozzáférniük a távoli cache-ekhez, hogy azok mind konzisztensek maradjanak.) A címet és a sínciklus jellegét "kidobja" két ciklusban az Address lábakon, a sor az első ciklusban megy ki, az oszlop pedig a második ciklusban, ahogy azt a 3-31.ábrán láthatjuk.

Az eredményre várva a CPU képes lehet folytatni a többi munkáját.Például, ha az utasítás előkészítése közben valamit nem talál a cache-ben, attól még végrehajthatja a már előkészített utaítást, amelyek hivatkoznak olyan adatra is, ami nincs benne egyik cache-ben sem. Így egyszerre több tranzakció is lehetséges az UPA-n keresztül. Az UPA képes egyszerre kezelni két egymástól független tranzakció sorozatot (ezek általában írások és olvasások), akár több megszakítással is. A központi vezérlő feladata mindezek nyomon követése és, hogy a leghatékonyabban sorrendben teljesítse a memória kéréseket.Amikor végül megérkezik az adat a memóriából, ez 8 byteban képes egyszerre lejönni, egy hibajavító kóddal a legnagyobb megbízhatóság érdekében. A tranzakció kérhet egy teljes cache blokkot, egy quadword-ot (8 byte), vagy akár még kevesebb byte-ot is. Minden beérkező adat az UDB-be kerül, amely pufferálja őket. Az UDB célja, hogy méginkább elhatárolja a CPU-t a memóriarendszertől, így azok aszinkronizáltan tudnak dolgozni. Például, ha a CPU-nak írnia kell egy szót vagy egy cache sort a memóriába, ahelyet, hogy várakozna az UPA elérésére már azonnal írhatja is az adatokat az UDB-be és hagyhatja, hogy az UDB később küldje az adatokat a memóriába. AZ UDB generálja és kontrollálja is a hibajavító kódot. Csak a teljesség kedvéért említjük, hogy az UltraSPARC II fenti leírása, mint ahogy a Pentium II leírása, rendkívül leegyszerűsített, de a művelet lényegét tartalmazza.

3.5.3 A picoJava II

Mind a Pentium II, mind az UltraSPARC II példa azon nagyteljesítményű CPU-kra, melyeket rendkívűli gyors PCK és munkaállomások építésére terveztek. Az emberek, amikor a számítógépre gondolnak, ilyen rendszerre össz-pontosítanak. Mindemellett létezik a számítógépek egy másik világa, amely tulajdonképp sokkal nagyobb: ez a beágyazott rendszereké. Ebben a fejezetben ebbe a világba látogatunk.

Talán enyhe túlzás lenne azt mondani, hogy minden 100&-nál drágább elektromos készülék számítógépet tartalmaz. Természetesen a TV, rádiótelefon, manager calculator, mikrohullámú sütő, kézikamera, videófelveő, lézernyomtató, riasztó, hallókészülék, elektronikus játékok és számos más készülék manapság számítógépes vezérléssel működik. A számítógépek ezekben az eszkö-zökben inkább az alacsony árhoz, mintsem a magas teljesítményhez lettek igazítva, ami más fejlesztéseket eredményezett , mint a magas szintű CPU-k, amikről szó

lesz.
Hagyományosana beágyazott processzorokat assembly nyelven programozták, de mivel a gépek egyre bonyolutabbaká váltak és a szoftveres hibáknak egyre komolyabb következményei lettek, így más megközelítések kerültek előtérbe. Különösen a Javának, mint programozási nyelvenek a használata beágyazott rendszereken lett vonzó, köszönhetően a viszonylag egyszerű programozhatóságok, kicsi kódméretek és a platformfüggetlenségnek.

A Java használat beágyazott alkalmazásának fő hátránya, hogy szükség van egy nagy szoftveres interpreterre, amely a Java fordító által adott JVM kódot hajtja végre, és hogy a Java fordítások és a fordítási eljárás igen lassú.

A Sun és más cégek úgy oldották meg a problémát, hogy terveztek és építettek egy olyan CPU chipet, amely alap utasításkészletként tartalmazta a JVM-et. Ez a megközelítés kombinálja a Java programnyelv használatának előnyeit, a hordozhatóságot, a Java fordító által készített JVM bináris kód kis méretét és a gyors végrehajtást speciális hardverrel. Ebben a részben egy modern Java alapú CPU architecktúrát nézünk meg, amit speciálisan beágyazott rendszerekre terveztek.

A CPU a Sun picoJava II, ami a Sun picoJava 701 chip alapja, bár Sun más cégektől is vásárolt terveket. A CPU egy egyszerű chip két busz interface-szel, az egyik a 64 bit széles memória busz, a másik a 32 bit széles PCI busz. Lásd 3.48 ábra. Ahogy a Pentium II és az UltraSPARC II-nél, a chipen osztott első szintű cache van, legfeljebb 16 KB az utasításoknak és legfeljebb 16 KB az adatoknak. Azonban a másik két CPU-nak nincsen második szintű cache-e, mivel az alacsony költség a kulcsfontosságú a beágyazott rendszerek tervezésénél. Lejjeb tárgyaljuk a Sun megvalósítását a picoJava II-re a microJava 701-et. A chip a jelenlegi mértékkel kicsinek mondható, csak 2 millió tranzisztorja van a magnak, illetve további 1,5 millió a két 16 KB-os cacheeknek.

A 3.48 ábráról három tulajdonság tűnik ki azonnal. Az első, a microJava 701 PCI buszt használ (33 MHz-t és 66 MHz-t is). Ezt a buszt az Intel fejlesztette ki magas szintű Pentium rendszerekhez, de ez a processzor független. A PCI busz használatának előnye az, hogy használa-

tával elkerülhető, hogy új buszt kelljen tervezni. Továbbá, hogy számos play in kártya kapcsolható hozzá. Bár a PCI kártyák létezése igen kis előny mobiltelefonok Építéséhez, Web Tv-k és egyéb fizikailag nagyobb eszközökhöz.

Másodszor ez a picoJava 701 rendszer alapállapotban tartalmaz flash PROM-ot. A lényeg, hogy egy eszközben nagyon sok,ha nem az összes program kell,hogy beépítve legyen. A flash PROM remek hely a program tárolására, tehát az interface léte a gépeken nagyon hasznos. Egy másik chip, amely adható a rendszerhez tartalmazza a soros és párhuzamos I/O interface-eket, amik a PC-n találhatók.

181

Harmadszor, a microJava 701-nek van 16 programozható I/O sora, amik kapcsolhatók gombokhoz, kapcsolókhoz és lámpák hoz a gépeken. Például a mikrohullámú sütőn gyakran vannak, számokat tartalmazó gombok illetve néhány egyéb gomb, amelyek közvetlenül a CPU chiphez kapcsolhatók. A közvetlenül a CPU-hoz kötött programozható I/O sorok szükségtelenné teszik a programozható I/O chipeket, ami az egész tervezést egyszerűbbé és olcsóbbá teszi. A chipen további három programozhat időzítő is van , ami szintén hasznos dolog, mivel az eszközök gyakran valós időben működnek. A microJava 701 a 316 tűlábas ipari szabványú BGA csomagban jön ki. Ezek közül 59 láb a PCI buszon van csatlakoztatva. A PCI buszt később részletesebben megvitatjuk a fejezetben. Másik 123 láb a memória buszé, amiből 64 kétirányú adatláb, valamint a különálló cím lábak. További lábakat használnak a vezérlésre (7), időzítőkhöz (3), megszakításhoz (11), tesztelésre (10), és a programozható I/O-ra (16). A többi láb közül néhányat használ a táp és a földelés, de a többit nem használja. A többi picoJava gyártó szabadon választhat különböző buszt, csomagot, stb. A chip rendelkezik számos egyéb hardver tulajdonsággal, mint például sleep mód az elem takarékosság miatt, chip megszakítás ellenőrző, és teljes támogatása az IEEE 1149.1JTGA teszt szabványoknak.

3.6 PÉLDÁK BUSZOKRA

A buszok azok a kapcsok, amelyek összetartják a számítógépes rendszert. Ebben a fejezetben megnézünk néhány ismertebb buszt: az ISA buszt, a PCI buszt és az Universal Serial Bus-t. Az ISA busz az IBM-PC egy csekély kibővítése. A visszamaradt összeférhtőség okánál fogva még mindig jelen van minden Intel based (alapú) PC-ben. Mindemellett minden gép rendelkezik egy második, gyorsabb busszal, méghozzá a PCI busszal. A PCI busz nagyobb kiterjedésű, mint az ISA és gyorsabb időt fut. Az Universal Serial busz egy mindinkább elterjedt I/O busz, kis sebességű perifériákkal, mint például az egér és a billenytyűzet. A következő részben sorjában véve megtekintjük az összes buszt.

3.6.1 Az ISA busz

Az IBM PC busz volt valójában a szabvány típus, amely a 8088 rendszeren alapult. Ugyanis majdnem minden PC klónt az árusok erről másoltak azzal a céllal, hogy megengedjék több már létező , harmadik partnerként I/O beszállását a rendszerük használatával. Az IBM PC busznak 62 jelsora van,beleszámítva 20-at a memória címek, 8-at az adatok miatt, és további egyet minden állításért: memória olvasás, memória írás, I/O olvasás és I/O írás. Ezek szintén kérelmi jelek és engedély a megszakításra, valamint a DMA használatára. Ez egy nagyon egyszerű
busz. Fizikailag a busz a PC alapjába van karcolva, körülbelül féltucat csatlakozóval, amelyek 2 cm távolságra vannak elosztva egymástól, s ezekbe lehet a kártyákat behelyezni.

***[182-185]

T 182-185

A kártyán volt egy lap ami illeszkedett a csatlakozóba.A lapocska mindkét oldalán 31 arannyal fedett szalag/csík volt, melyek a csatlakozóval való elektronikai kontaktust biztosítják. Amikor az IBM bevezette a 80286 alapú PC/AT-t, nagy problémával állt szemben. Ha egy teljesen újfajta 16 bit-es adatcsatornát tervezett volna, sok potenciális ügyfél vonakodott volna attól, hogy megvegye, mivel az a sokfajta PC „bedugós rendszerű" kártya, melyeket harmadrangú eladók terjesztettek, nem lettek volna üzemképesek az új adatcsatornával. Másrészt a PC adatcsatornához és annak 20 cím- és 8 adatvonalához nem használták volna ki a 80286 azon képességét, hogy 16 Mb-s memóriát címezzen és 16 bites szavakat adjon át.

Azt a megoldást választották, hogy kibővítik a PC adatcsatornát. A PC „bedugós rendszerű" kártyáknak 62 tűs NYÁK-csatlakozójuk van, mely nincs olyan hosszú, mint az áramköri lap. A PC/AT megoldás azt jelentette, hogy egy második NYÁK csatlakozót helyeznek az áramköri lap aljára, mely közvetlenül a fő csatlakozó mellett van és olyan AT áramkört terveznek, mely mindkét fajta áramköri lappal dolgozik. Az alapötletet a 3-49-es ábra illusztrálja.

A PC/AT adatcsatornán elhelyezkedő második csatlakozó 36 vonalat tartalmaz. Ezek közül 31 vonal több cím-, adat- és programmegszakítási-vonalat, valamint több DMA csatornát lát el. A többi a 8 és 16-bites átvitel különbségeivel foglalkozik.

Amikor az IBM piacra bocsátotta a PC és PC/AT utódát, a PS/2 szériát, egy teljesen új tervezetet hozott létre. Ez a döntés részben technikai alapokon nyugodott (a PC adatcsatorna ekkorra már teljesen elavult), de részben annak a törekvésnek volt köszönhető, hogy gátolni akarták azokat a cégeket, melyek PC utánzatok készítésével

foglalkoztak, és a piac igen nagy részét uralták. Ezért látta el az IBM a közép- és felsőkategóriájú PS/2 gépeket egy adatcsatornával,a Microchannellel, mely úttörő jellegű volt, és melyet egy ügyvédekből álló „szabadalonfal” védett.

A számítógépipar többi társasága úgy reagált erre a lépésre, hogy felállította saját szabványát, az ISA adatcsatornát, mely alapjában véve 8.33 MHz-en futó PC/AT. Ennek az intézkedésnek az az előnye, hogy megtartja a meglévő gépekkel és kártyákkal való kompatibilitást. Ez szintén egy olyan adatcsatornán alapul, mely gyártását az IBM számtalan cégnek szabadon engedélyezett annak érdekében, hogy biztosítsa, hogy a lehető legtöbb harmadrangú cég gyártsjon kártyát az eredeti PC-hez. Ez a döntés balul ütött ki az IBM számára. Minden Intel alapú PC még mindig rendelkezik ezzel az adatcsatornával, bár emellett még általában megtalálható bennük legalább egy más jellegű adatcsatorna is. A részletes leírás Shanley és Andersen 1995-ben kiadott munkájában található.

A későbbiekben az ISA-t néhány új elem (pl. szimultán programfutás) beiktatásával 32 bite-ra bővítették Az új adatcsatorna neve EISA (bővített ISA) lett. Ehhez viszont igen kevés áramköri lapot gyártottak.

3.6.2 A PCI adatcsatorna

Az eredeti IBM PC-n a legtöbb alkalmazás szöveges módban működött. A Windows bevezetésével fokozatosan grafikus használói csatolóegységeket kezdtek használni.Ezek az alkalmazások közül egyik sem terhelte meg túlságosan az ISA adatcsatornát. A helyzet azonban idővel gyökeresen megváltozott annak köszönhetően, hogy sok olyan alkalmazás (különösképpen a multimédia játékok) jelentek meg, melyek a számítógépet teljes képernyős , teljes videók bemutatására használták.

Végezzünk egy kis számítást. Van egy 1024x768-as képernyőnk, melyet élethű szinekben megjelenő (3 byte/képelem) mozgó képek bemutatására használunk. Egy képernyő 2.25 Mb-nyi adatot tartalmaz. A mozdulatok kifinomítására másodpercenként legalább 30 kép szükséges, másodpercenként 67.5 MB-nyi adat szolgáltatására. A helyzet tulajdonképpen még rosszabb, mivel egy videó hard disk-ről, CD-ROM-ról vagy DVD-ről való bemutatása esetén az információnak a meghajtóról az adatcsatornán keresztül kell eljutnia a memóriába. Ezután a bemutatáshoz az információ újra keresztül kell hogy menjen az adatcsatornán és csak ezután jut el a grafikus adapterhez. Már magához a videóhoz egy másodpercenkénti 135 MB-os sávszélességű adatcsatornára van szükség, nem számolva a sávszélességet ami a CPU és más egységek üzemeltetésének feltétele.

Az ISA adatcsatorna maximálisan 8.33 MHz-es sebességgel működik és ciklusonként 2 byte-ot tud átadni 16.7 MB/sec.-mal. Az EISA adatcsatorna ciklusonként 4 byte-ot tud mozgatni, 33.3 MB/sec-es sebességgel. Tehát ezek közül egyik sem alkalmas a teljes képernyős videó bemutatására.

1990-ben Intel előrelátta ezt és egy új adatcsatornát tervezett, mely átviteli sebessége sokkal nagyobb, még az EISA adatcsatornáénál is. Az új adatcsatornát PCI-nak nevezték. Az új adatcsatorna népszerűsítésének érdekében az Intel szabadalmaztaa PCI-t, majd az összes szabadalmat közös tulajdonba adta, így bármelyik társaság készíthetett hozzá perifériákat, anélkül, hogy szabadalmi díjat kellene fizetnie. Az Intel emellett létrehozott egy ipari társulatot, a PCI Special Interest Group-ot, mely a PCI adatcsatorna jövőjét irányította. Ezeknek az intézkedéseknek a következményeként a PCI adatcsatorna igen népszerű lett. Elméletileg a Pentium megjelenése óta minden Itel-alapú számítógépnek PCI

adatcsatornája van, és sok más számítógépnek is. A Sunnak még egy olyan UltraSPARC verziója is van, ami PCI adatcsatornát használ, ez az UltraSPARC IIi. A PCI adatcsatornáról részletes leírást Shanley és Anderson, 1995b; valamint Solari és Willse 1998, munkájában találhatunk.

Az eredeti PCI adatcsatorna ciklusonként 32 bitet tud átvinni és 33Mhz-en fut, maximálisan 133 MB/sec sávszélességgel. 1993-ban jelent meg a PCI 2.0, 1995-ben pedig a PCI 2.1. A PCI 2.2 alkalmas mobil számítógépekhez (főleg energiamegtakarításra használják). A PCI adatcsatorna 66 MHz-en fut és 64-bit átvitelét teszi lehetővé, 528 MB/sec.-os sávszélességen. Ez a kapacitás lehetőséget nyújt a teljes képernyős, teljes mozgású videoképek feldolgozására (feltételezve azt, hogy a disk és a rendszer többi része készen áll (szabad) a feladatra). Mindenesetre a PCI nem okozza a rendszer fennakadádát.

Habár az 528MB/sec igen gyorsnak tűnik, mégis felmerül vele kapcsolatban két probléma. Először is nem megfelelő tároló-adatcsatornának, másodszor pedig nem összeegyeztethető a régi ISA kártyákkal. Intel szerint az a megoldás, hogy olyan számítógépeket tervezzenek, melyek 3 vagy több adatcsatornával rendelkeznek (3-50 ábra). It látható, hogy a CPU kommunikálhat a fő memóriával egy különleges tároló adatcsatorna segítségével, valamint, hogy egy ISA adatcsatorna összeköthető a PCI adatcsatornával. Ez a megoldás minden igénynek megfele, ezért tulajdonképpen minden Pentium II számítógép ilyen felépítésű.

A két fő összetevője ennek a felépítésnek a két híd-chip (amit az INTEL gyárt, ezért érdekelt ebben a tervben). A PCI híd összekapcsolja a CPU-t, a memóriát és a PCI adatcsatornát. Az ISA híd a PCI adatcsatornát és az ISA adatcsatornát köti össze, valamit egy vagy két IDE lemezmeghajtót támogat. Majdnem minden Pentium II rendszer egy vagy több szabad PCI csatlakozóval rendelkezik, ami új nagy gyorsaságú perifériák, valamint a lassú perifériákkal való összekötés érdekében egy vagy több ISA csatlakozó hozzákapcsolását teszi lehetővé.

A 3-50 ábra előnye, hogy a CPU-nak különösen magas a sávszélessége a memória irányába, a saját memóriájának használata a PCI adatcsatornának igen nagy sávszélességet biztosít a gyors perifériák, mint pl. SCSI lemezmeghajtók, grafikus stb. irányába, valamint a régi ISA kártyák használata is lehetséges. Az ábrán az USB doboz az Universal Serial Bus (univerzális soros adatcsatorna) rövidítése, erről az adatcsatornáról a fejezet további részében esik szó.

Bár az ábrán egy PCI és egy ISA adatcsatornával rendelkező rendszert illusztráltunk, többszörös adatcsatorna működtetése is lehetséges. Használhatóak pl. a PCI-to-PCI híd-chipek, melyek két PCI adatcsatorna összekötésére szolgálnak, így a nagyobb rendszerek két vagy több, egymástól független PCI adatcsatornával rendelkezhetnek. A rendszerbe két vagy több PCI-to-ISA híd-chip beépítése is lehetséges, mely többszörös ISA adatcsatornát tesz lehetővé.

Jó lenne, ha csak egy fajta PCI kártya létezne, de sajnos nem ez a helyzet. A PCI kártyák feszültség , szélesség és időzítés függvényében változnak. A régebbi típusú számítógépek gyakran 5 volttal, az újabbak pedig a 3.3 volttal működnek, ezért a PCI adatcsatorna mindkettőt támogatja. A csatlakozók között az egyetlen kis eltérés, hogy van rajtuk két kis műanyag, melynek az a funkciója, hogy megakadályozza, hogy az 5 voltos kártyát a 3.3 voltos PCI adatcsatornába illesszék be, vagy fordítva. Szerencsére léteznek univerzális kártyák is, melyek mindkét feszültséget támogatják, és mindkét fajta csatlakozóba beilleszthetőek. A feszütségváltozatok mellett a kártyáknak több fajtája létezik a bitek száma szerint is, 32-bites és 64-bites verziót különböztethetünk meg. A 32-bites kártyák 120 tűs csatlakozóval rendelkeznek, a 64-

bitesek pedig a 120 tűs csatlakozó mellett további 64 tűs csatlakozóval van ellátva, hasonlóan ahhoz, ahogy az IBM PC adatcsatornát 16 bitre bővítették (lásd 3-49 ábra). Az a PCI adatcsatorna, mely a 64-bites kártyákat támogatja 32-bites kártyákkal is üzemel, fordítva viszont nem. Végezetül a PCI adatcsatornák és kártyák mind 33, mind 66 MHz-en futnak. Két lehetőség van, vagy a tápfeszültségre, vagy a földeléshez kötünk egy vezetéket. A csatlakozók mindkét esetben egyformák. A PCI adatcsatorna szinkron adatátvitellel működik, mint minden PC adatcsatorna, kezdve az eredeti IBM PC-vel. A PCI adatcsatornán minden tranzakció egy mester (hívatalos nevén **initiator**, kezdeményező)és egy szolga (hívatalos nevén **target,** tárgy/cél) között jön létre. Azért, hogy a PCI tűs csatlakozóba kevesebb tű legyen , a cím- és adatvonalak multiplexek. Így a PC kártyákon csak 64 pinre van szükség a cím- és adatjelhez, annak ellenére, hogy a PCI a 64-bites címeket és adatokat támogatja.

A multiplex cím- és adatvonalak a következőképpen működnek. Az olvasás során az első ciklusban a mester a címet az adatcsatornára teszi. A második ciklusban a mester leveszi a címet és az adatcsatorna megfordul, így a szolga számára elérhetővé válik. A harmadik ciklusban a szolga kiadja a kért adatot. Az írás során az adatcsatornát nem kell megfordítani, mivel mind a címet, mind az adatot a mester biztosítja.

***[186-189]

Azonban a minimális tranzakció 3 ciklusos. Ha a slave nem képes 3 cikluson belül válaszolni, akkor beállítja várakozó állapotba.

PCI Bus Irányítás

Ahhoz, hogy használuk a PCI bus-t , egy eszköznek először kapcsolatba kell
lépnie vele.A PCI Irányítás egy központosított bus irányítót használ, ahogy azt
a 3-51. ábra mutatja. A legtöbb konstrukcióban a bus irányító az egyik Híd chip-
be van beleépítve. Minden PCI eszköznek 2 külön vonala fut az irányítóba. Az
egyik vonalat, a REQ#-et, arra használa, hogy hívja a bus-t. A másik vonalat, a
GNT#-t, arra, hogy fogada a bus engedélyezéseit.

PCI arbiter: PCI irányító

PCI device: PCI eszköz

3-51.ábra: A PCI bus központosított bus irányítót használ.

Ahhoz, hogy hívjuk a bust egy PCI eszköz (beleértve a CPU-t) lefoglalja a REQ# vonalat, és vár, amíg az ellenőrzi az irányító által lefoglalt GNT# vonalat.
Amikor az eredmény meg van, az eszköz a következő ciklusban használhatja a
bus-t. Az irányítóáltal használt algoritmus nem értelmezett a PCI előírás által.
Round robin irányítás, priority irányítás és más rendszerek engedélyezettek. Egy
jó irányító 'igazságos', és nem engedi az eszközöket az örökkévalóságig várni.

Egy bus engedélyezés egy tranzakcióhoz vonatkozik, jólehet a tranzakció hossza elméletileg határtalan. Ha egy eszköz egy második tranzakciót akar el-
indítani, és egy másik eszköz nem hívja a bus-t, akkor az újra elindulhat, noha a
tranzakciók között be kell iktatni egy üres ciklust. Ám speciális körülmények
között a bus-ért folytatott küzdelem hiányában egy eszköz oda-vissza tranzakci-
ókat hajthat végre üres ciklusok beiktatása nélkül. Ha egy bus master egy na-
gyon hosszú tranzakciót hajt végre, és más eszközök is hívják a bus-t, az

irá-
nyító neghálni tuda a GNT# vonalat. Az elterjedt bus master vár, és figyeli a
GNT# vonalat, így amikor észleli a negációt, a következő ciklusban felszaba-
dítja a bust.Ez a rendszer lehetővé tesz nagyon hosszú tranzakciókat (melyek
eredményesek) amikor csak egyetlen jelölt bus master van, de ez mégis gyors
reagálást tesz lehetővé a versenyben lévő eszközöknek.

PCI Bus Jelek

A PCI bus-nak számos utasító ele van, ahogy az a 3-52(a) ábrán látható, és
számos opcionális jele van, ami a 3-52(b) ábrán látható. A többi 120 vagy 184
lábat az áramhoz, a földöz, és különféle összefüggő funkciókhoz használják,
amelyeket itt nem soroltunk fel. A Master (kezdeményező), és a Slave (cél) osz-
lopok tájékoztatnak arról, hogy normál tranzakcióknál ki foglalja le a jelet. Ha a
jel egy eltérő eszköz által van lefoglalva (pl.: a CLK), akkor mindkét oszlop üres.

Most nézzük meg röviden a PCI bus jelek mindegyikét. Az utasító jelekkel
(32 bit) kezdjük, utána folytatjuk az opcionális jelekkel (64 bit). A CLK jel irá-
nyítja a bus-t. A többi jel nagyrésze szinkronban van vele. Az ISA bus-szal
ellentétben a PCI bus tranzakciója az órajel lefutó élében kezdődik, ami a ciklus
közepén van, semmint az elejénél.

A 32 AD jelek a címnek és az adatnak van (32 bites tranzakciónál). Rendsze-
rint az 1-es ciklus alatt a cím, a 3-as ciklus alatt pedig az adat van hívva. A PAR
jel megfelelő bitszámú az AD-nek. A C/BE# jelet két különböző dologra hasz-
nálják. Az 1-es ciklus alatt, amely a bus parancsokat tartalmazza (1 szó beolva-
sása, blokk beolvasása, stb.). A 2-es ciklus alatt, amely 4 bitnek a

bittérképét tar-
talmazza, mely informál arról, hogy a 32 bites szó melyik bájtjai érvényesek. A
C/BE#-t írni, és olvasni is lehet bármely 1, 2 vagy 3 bájtot, valamint egy egész
szót is.

A FRAME# jel a bus master által van lefoglalva, hogy bus tranzakciókat kezd-
hessen. Ez informálja a slave-et, hogy a cím, és a bus parancsok érvényesek e. Az írás alatt általában az IRDY# is le van foglalva egy időben a FRAME#-el.Ez
közli a masterrel, hogy kész bejövő adatok fogadására. Olvasáskor az IRDY# később lesz lefoglalva akkor, amikor az adat már a bus-on van.

Az IDSEL jel összefüggésben van azzal a ténnyel, hogy minden PCI eszköz-
nek egy 256 bájt konfigurációs hellyel kell rendelkeznie, amit más eszközök is
el bírnak olvasni (az IDSEL lefoglalásával). Ez a konfigurációs hely az eszkö-
zök tulajdonságait tartalmazza. Néhány operációs rendszer a Plug 'n Play tulaj-
donsággal a konigurációs helyet arra használja, hogy mely eszközök használják
a bus-t.

Most a slave által lefoglalt jeleket nézzük. Ezek közül az első a DEVSEL#,
mely közli, hogy a slave észlelte a címét az AD vonalakon, és kész részt venni
a tranzakcióban. Ha a DEVSEL# egy bizonyos időhatáron belül nem lesz lefog-
lalva, akkor a master szünetet tart és úgy veszi, hogy a címzett eszköz hiányzik,
vagy hibás.

A második slave jel a TRDY#, melyet olvasásnál a slave azért foglal le, hogy
az adatok az AD vonalakon vannak e, és írásnál az adatok befogadásának felké-
szűltségéről informáljon.

A másik 3 jel hibajelzésre szolgál. Az első a STOP#, melyet a slave akkor használ, ha valami tragikus dolog történik, és le akarja állítani az aktuális tran-
zakciót. A következő a PERR#, mely az adatok prioritási hibájáról

informál az
első ciklus alatt. Olvasáshoz a master, íráshoz a slave foglalja le. A vevőn múlik,
hogy az kellő műveletet hajtson végre. És végül a SERR# jel, amely a címzési-
és rendszerhibákról informál.

Lines: vonalak Signal: jel	Description: utasítás
CLK:	Óra (33 MHz vagy 66MHz)
AD:	Összetett cím-, és adatvezetékek
PAR:	Cím vagy adat bitmegfelelések
C/BE:	Bus parancs-, bittérkép bájtengedélyezésekhez
FRAME#:	Az AD és a C/BE jelek foglaltságát jelzi

Signal: jel	Description: utasítás
IRDY#:	Olvasás: a master elfogadja; írás: adat prezentálás
IDSEL:	Konfigurációs hely kiválasztása a memória helyett
DEVSEL#:	A Slave először dekódólja a címét, majd figyel
TRDY#:	Olvasás: adat prezentálás; írás: slave elfogadja
STOP#:	A slave azonnal le akarja állítani a tranzakciót
PERR#:	A vevő által észlelt paritási hiba
SERR#:	Adatparitási hiba vagy rendszerhiba van észlelve
REQ#:	Bus irányítás: a bus irányításának kérelme
GNT#:	Bus irányítás: a bus irányításának engedélyezése
RST#:	A rendszer, és az összes eszköz újraindítása

(a)

Sign: jelzés	Description: utasítás
REQ64#:	64 bites tranzakciókhoz hívják
ACK64#:	64 bites tranzakciókhoz garantálva van a jóváhagyás
AD:	További 32 bitnyi cím vagy adat
PAR64:	32 bitnyi extra adat/cím paritás
C/BE#:	További 4 bit bájtengedélyezéshez
LOCK:	A bus lezárása összetett tranzakciók engedélyezéséhez
SBO#:	Távoli chace-ra való találás (multiprocesszorokhoz)
SDONE:	Ellenőrzés vége (multiprocesszorokhoz)
INTx:	Megszakítás kérése
JTAG:	IEEE 1149.1 JTAG teszt jelek
M66EN:	A tápeszültséggel, vagy a földdel van összeköttetésben (66 MHz vagy 33 MHz)

(b)

3-52. ábra: (a)Utasító PCI bus jelek. (b)Opcionális PCI bus jelek.

A REQ#, és a GNT# jelek bus irányítást végeznek. Ezek az aktuális bus master által nincsennek lefoglalva, inkább az az eszköz által, amelyik bus master akar lenni. Az utolsó utasító jel az RST#, amelyet a rendszer blokkolásra használ akkor, ha a felhasnáló megnyomja a RESET gombot, vagy más rendszer eszköz
Fatal errort észlel. Ezt a jelet észlelve blokkolja az összes eszközt, és újra boot-olja a számítógépet.

Most elérkeztünk az opcionális jelekhez, amalyek közül a legtöbb összefüg-
gésben van a 32 bitről 64 bitre történő kibővítéssel. A REQ64# és az ACK64#
jelek lehetővé teszik a masternek, hogy engedélyt kérjen egy 64 bites tranzak-
ció lebonyolításához, és engedje a slave-et elfogadni az külön-külön. Az AD,
PAR64, és a C/BE# jelek a megelelő 32 bites jelek kiterjesztései.

A következő 3 jel nincs kapcsolatban a 32 bittel, ellentétben a 64 bittel, de a
multiprocesszor rendszer igen, valami, amit a PCI board-ok nem igényelnek,
hogy támogassák. A LOCK jel engedélyezi a bus-nak, hogy el legyen zárva a
multiplex tranzakciók elől. A kövezkező kettő a bus kereséssel van kapcsolat-
ban, hogy fenntartsa a cache összefüggést.

A INTx jelek megszakítás kérérsére vannak Egy PCI kártya több mint 4 elkü-
lönített logikai eszközt tartalmazhat, és mindegyiknek lehet saját megszakítást
kérő vonala. A JTAG jelek a JEEE 1149.1 JTAG tesztfolyamatához vannak. És
végül a M66EN jel, ami lehet magas vagy alacsony, hogy beállítsa az óra sebes-
ségét. Nem szabad megváltoztatni rendszeroperációk alatt.

PCI Bus Tranzakciók

A PCI bus tényleg nagyon egyszerű (a buszokhoz képest). Hogy jobban értsük

, figyeljük meg a 3-53. Ábrát, a számláló diagrammot. Itt egy olvasás folyamatot
látunk, melyet egy üres ciklus követ, amit pedig egy író folyamat ugyanazzal a
bus master-rel.
Amikor a leutó éle jön el az órajelnek a T1 alatt, a master a memória címet az
AD-re, a bus parancsot pedig a C/BE#-re rakja. Ezután utasítja a FRAME#-et,
hogy elkezdje a bus tranzakciót.
A T2 alatt a master készenlétben tartja a címbust, hogy megforduljon, és ké-
szen álljon a slave-nnek ahhoz, hogy működtesse a T3 alatt. A master a C/BE#-t
is megváltoztatja, hogy jelezze, melyik címzett bájtot akarja engedélyezni (azaz beolvasás).
A T3 alatt a slave utasítja a DEVSEL#-t, így a master tudja, hogy megkapta a
címet, és tervezi a választ. Ugyancsak az AD vonalakra teszi az adatot,és utasitja
a TRDY#-t, hogy közölje a master-ral,hogy kész van. Ha a slave nem képes elég
gyorsan válaszolni, akkor még utasítja a DEVSEL#-t, hogy közölje az
állapotát, de a TRDY#-t tartsa negálva, míg az képes az adatot kivinni. Ez a folyamat beil-
leszthet egy vagy két várakozó állást.
Ebben a példában (és gyakran a valóságban) a következő ciklus üres. A T5-öt
elkezdve láthatjuk ugyanazt a mastert elindítani egy írást. Rendszerint a címnek
és a parancsnak a bus-ra való felhelyezésével indul. Csak most, a második cik-
lusban közli az adatot. Azóta, hogy ugyanaz az eszköz vezeti az AD vonalat
nincs szükség a megfordítás ciklusra. A T7-ben a memória elfogadja az adatot.

Az Univerzális Serial Bus

A PCI bus jö nagysebességű perifériák számítógéphez való csatolására, de túl
drága, hogy minden alacsony sebességű I/O eszköznek külön PCI
interface-ja legyen, mint a billentyűzetnek, vagy az egérnek.

Történelmileg minden szten-
derd I/O eszköz egy speciális módon van csatlakoztatva a számítógéphez.
Né-
hány szabad ISA és PCI bővítőkártya hely új eszköz hozzáadását teszi lehetővé.

***[190-193]

T190-193 h938455

Sajnos, ez az elrendezés a kezdetektől fogva tele van problémákkal. Például gyakran minden új I/O egység a saját ISA vagy PCI kártyájával kerül forgalomba. Gyakran a felhasználó felelőssége, hogy a kapcsolókat és jumpereket (hardverkapcsolókat) beállítsa a kártyán és biztosítsa a kártyák összeférhetőségét. Azután a felhasználónak ki kell nyitnia a gépdobozt, óvatosan behelyezni a kártyát, becsuknia a dobozt és újraindítania a számítógépet. Sok felhasználó számára ez a folyamat nehéz és hibalehetőséget biztosít. Továbbá az ISA és PCI slot-ok (nyílások) száma nagyon korlátolt (egy vagy kettő típusonként). A Plug `n Play (bedugós rendszerű) kártyák kiküszöbölik a hardverkapcsolók beállítását, de a felhasználónak még mindig ki kell nyitnia a számítógépet, hogy belehelyezze a kártyát és a busz nyílások száma még mindig korlátolt.

Foglalkozva ezzel a problémával, a 1990-es évek közepén hét vállalat (Compaq, DEC, IBM, INTEL, Microsoft, NEC, Northern Telecom) képviselői összefogtak, hogy egy jobb módszert tervezzenek kis sebességű I/O egységek számítógéphez csatlakoztatására. Azóta más vállalatok százai csatlakoztak hozzájuk. Az így kialakult szabványt **USB**-nek **(Universal Serial Bus; Univerzális Soros Busz)** nevezik és ez már széles körben elterjedt a személyi számítógépek között. Alapos leírás (Anderson, 1997; Tan, 1997).

A vállalatok, amelyek eredetileg kigondolták az USB-t és elkezdték a projektet, céljaiból néhány a következő:

1. A felhasználóknak ne kelljen beállítaniuk a kapcsolókat vagy a hardverkapcsolókat az áramköri lapokon vagy az egységeken.
2. A felhasználóknak ne kelljen kinyitniuk a gépdobozt, hogy installálják az új I/O egységeket.
3. Csak egyetlen egy féle kábelre van szükség, ami jó minden egység csatlakoztatására.
4. Az I/O egységeknek a kábelekről kellene az áramot kapniuk.
5. 127 egységig lehessen csatlakoztatni egyetlen egy számítógéphez.
6. A rendszernek támogatnia kellene a valós idejű egységeket (pl.: hang, telefon).
7. Az egységeknek installálhatóaknak kellene lenniük amíg a számítógép fut.
8. Ne legyen szükség az újraindításra egy új eszköz felinstallálása után.
9. Az új busz és I/O egységeknek olcsó legyen az előállításuk.

Az USB megfelel mindegyik célnak. Ezt kis sebességű egységekhez tervezték, mint a billentyűzetek, egerek, kamerák, scannerek (képletapogató), digitális telefonok és így tovább. A teljes USB sávszélessége 1.5MB/sec, ami elég tekintélyes számú egységhez. Ezt az alacsony korlátot a költségek lenntartása miatt választották.

Az USB rendszer egy **root hub-**ból **(gyökérközpont)** áll , ami a központi buszba csatlakozik (ld.: 3-53. ábra). Ennek a hub-nak csatlakozói (sockets) vannak a kábelekhez, amik kapcsolódhatnak I/O egységekhez vagy növekvő hub-okhoz, hogy több csatlakozót biztosítsanak, és így az USB rendszer topológiája egy fa a gyökerével, ami a gyökér hub-nál van, a számítógép belsejében. A kábeleknek különböző csatlakozóik vannak a hub végen és az egységek végein, hogy megakadályozzák az embereket, hogy véletlenül két hub-ot összekapcsoljanak.

A kábel négy fajta huzalból áll: kettő az adatoké, egy az áramé (+5volt) és egy

a földelésé. A jelzőrendszer 0-át közvetít feszültségátmenetként és 1-et feszültségátmenet hiányában, így a 0-ák hosszú futásai generálnak egy szabványos impulzusáradatot.

Amikor egy új I/O egységet bedugnak, a gyökér hub észleli ezt az eseményt és megszakítja az operációs rendszer futását. Ezután lekérdezi az eszközt, hogy kitalálja vajon milyen és mennyi sávszélességre van szüksége. Ha az operációs rendszer szerint van elég sávszélesség az egység számára, akkor kijelöl az új egységnek egy egyedüli címet (1-127) és letölti ezt a címet más információkat, a regiszterek konfigurálására az egységeken belül. Ezen a módon, az új egység szabadon (on-the-fly) hozzáadható, anélkül, hogy bármilyen felhasználói konfigurálásra szükség lenne és anélkül, hogy új ISA vagy PCI kártyát kellene installálni. A nem beállított kártyák a 0 címmel indulnak, így ezek címezhetőek. A kábelrendszer egyszerűsítésére sok USB egység beépített hub-okat tartalmaz, hogy elfogadja a pótlólagos USB egységeket. Például a monitor két hub csatlakozójához a bal- és jobboldali hangszórót lehet hozzáilleszteni.

Logikailag az USB rendszert egy bit csatornarendszernek lehet tekinteni, ami a root hub-tól az I/O egységekhez. Minden egyes egység feloszthatja a saját bit csatornáját általában 16 alcsatornára a különböző típusú adatok részére (pl.: audio, video). Mindegyik csatornán és alcsatornán adatok áramolnak a root hub-tól az eszközhöz vagy a másik irányba. Két I/O egység között nincs forgalom.

Pontosan minden 1.00±5.00 msec-ban a root hub egy új **frame**-et közvetít, hogy minden egységet időben összhangban tartson. A frame össze van kapcsolva egy bit csatornával és csomagokat (kb.: 1000bit) tartalmaz, amelyek közül az első a gyökér hub-tól az egységhez vezet. A frame további csomagjai is ebben az irányban futnak vagy az egységtől visszakerülhetnek a root hub-ba. Négy frame szerkezetét mutatja a 3-54. ábra.

A 3-54. ábrán a 0-ás és a 2-es frame-ben nincs munka, így egy **SOF** (Start of Frame; frame kezdete) csomagra van csak szükség. Ezt a csomagot minden egység megkapja. Az 1-es frame egy lekérdező, vagyis például a scannertől kéri , hogy adja vissza azokat bit-eket (bitképet), amit a képen talált a scannelés alatt. A 3-as frame adatszállítást tartalmaz valamilyen egységekhez, pl. nyomtatóhoz.

Az USB 4 féle frame-et támogat: irányító (control),egyidejű (isochronous), tároló (bulk), megszakító (interrupt). Az irányító frameket az egységek konfigurálására alkalmazzák, hogy parancsokat adjanak nekik és lekérdezzék az állapotukat. Az egyidejű frameket a valós idejű egységekhez alkalmazzák, mint a mikrofonok, hangszórók és telefonok, amiknél nagyon szükséges az adatok pontos időtartománybeli küldése és fogadása. Nagyon előre látható késleltetést tartalmaz, de biztosítja, hogy ne legyen adatismétlés a hibás eseménysorokban. A tároló frame-ek az eszközökre vagy az eszközökből történő nagy mennyiségű adatátvitelhez kellenek, amelyeknél nem szükséges a valós idejűség, mint pl. a nyomtatóknál. Végül, a megszakító frame-ekre, azért van szükség, mert az USB nem támogatja a megszakításokat. Például ahelyett, hogy a billentyűzet egy megszakítást okozna, amikor egy billentyűt lenyomunk, inkább az operációs rendszer lekérdezi a billentyűzetet minden 50Msec-ban, hogy összegyűjtse az összes függésben lévő kezelőbillentyűt.

Egy frame egy vagy több csomagból áll, lehetőleg néhány egy irányban áll. Négy fajta csomag létezik: token (vezérjel), data(adat), handshake (kézrázás) és special (speciális). A token csomagok a gyökértől az egységhez futnak és a rendszervezérlés a feladatuk. A SOF, IN és OUT csomagok a 3-54.ábrán token

csomagok. A SOF (Start of Frame; frame kezdete) csomag az első minden frame-ben és a frame kezdetét jelzi. Ha itt nincs munka, akkor a SOF csomag az egyetelen a frame-ben. Az IN token csomag egy lekérdező (poll), amely az egységet kérdezi le, hogy visszakapja a pontos adatait. Az IN csomag mezői elárulják, hogy melyik bit csatornát kérdezi le éppen, így az egység tudja, hogy milyen adattal térjen vissza (ha vannak többszörös streamjei (adattár?)). Az OUT token csomag közli, hogy adat fog érkezni az egységtől. A token csomag egy negyedik típusa , a SETUP (nincs jelölve az ábrán), konfigurálásra használható.

A token csomagon kivűl 3 más fajta létezik még. Ezek a DATA (adatok; a 64byte feletti információk más módú adatátvitelére használják), handshake és a special csomagok. Az data csomag szerkezetét a 3-54. ábra mutatja. Ez egy 8-bites egyidejűsítő mezőből, egy 8-bites csomag típusból (PID), a payload-ból (hasznos teher) és egy 16-bites **CRC** (cycle redundancy check)-ből áll, mely észleli a hibákat. Három fajta handshake csomag van meghatározva : ACK ((acknowledgement; nyugtázás) az előző adatcsomag megfelelően megérkezett), NAK ((negative acknowledge) a CRC hibát észlelt) és SFALL (kérem várjon- éppen el vagyok foglalva).

Most nézzük meg újra a 3-54. ábrát. Minden 1.00Msec-ban egy frame-et el kell küldeni a gyökér hub-tól, meg ha esetleg nincs is munka. A 0-ás és a 2-es frame csak egy SOF csomagot tartalmaz, jelezve, hogy ott nincs munka. Az 1-es frame egy lekérdező, így ez egy SOF és egy IN csomaggal indul ki a számítógéptől az I/O egységhez, melyet egy DATA csomag követ az I/O egységtől a számítógéphez. Az ACK csomag jelzi az egységnek, hogy az adat megfelelően megérkezett. Hiba esetén, egy NAK csomagot küldenének vissza az egységhez és a csomagot a bulk (tároló) adathoz küldenék vissza (de nem a isochronous adathoz). A 3-as frame hasonló szerkezetű az 1-essel, kivéve hogy most az adatáramlás a számítógéptől az egységhez tart.

3.7. Csatlakoztatás

Egy jellegzetes kis- és középméret közötti számítógép rendszer tartalmaz egy CPU chip-et (áramköri lapka), egy memória chip-et és néhány I/O vezérlőt, amelyek mind egy busszal vannak összekapcsolva. Már részben tanultunk a memóriákról, a CPU-król, és a buszokról. Most itt az ideje, hogy megnézzük a “rejtvény” utolsó darabját, az I/O chip-eket. Ezek azok a chip-ek, amelyeken keresztül kommunikál a számítógép a külvilággal.

3.7.1 I/O chipek

Számos I/O chip már rendelkezésre áll és vannak újak, amelyeket bármikor bevezethetnek. Az egyszerű chipek tartalmaznak UART-kat, USART-kat, CRT vezérlőket, lemez vezérlőket és PIO-kat. Az **UART (Universal Asynchronous Receiver Transmitter; univerzális aszinkron adó-vevő)** egy olyan chip, amely be tud olvasni egy byte-nyi adatot az adatbuszból és kiírni egyszerre egy bitet a terminálhoz vezető soros vonalra, vagy adatot tud beolvasni a terminálról. Az UART-k általában engedélyezik a változó sebességet; a 5-től 8 bitig terjedő karaktert; 1, 1.5 vagy 2 megállító biteket; és biztosítja a páros, páratlan, vagy paritás nélküli tulajdonságot minden programirányítás alatt. **USART-k (Universal Synchronous Aszinchronous Receiver Transmitters; univerzális szinkron/aszinkron adó-vevő)**

tudják kezelni a szinkronátvitelt különféle protokollok felhasználásával, valamint végrehatják az összes UART funkciót is.

3-53. ábra Példa egy 32 bites PCI busz tranzakcióira. Az első három ciklus egy olvasási művelet, aztán egy inaktív ciklus következik, és aztán három ciklus egy írási művelet.

3-54. ábra Az USB root hub (gyökérközpont) minden 1.00 msec-ban egy frame-et küld el.

frame : adatátvitelnél a folytonos bitsorozatokkezdő és záró flagsorozatok közé vannak zárva
idle : inaktív

***[194-197]

PIO Chips

A 3-55. ábrán látható Intel 8255A chip egy tipikus PIO (Parallel Input/Output: Párhuzamos Bemenet/Kimenet) chip. 24 I/O vonala van, amivel csatlakozni tud bármilyen TTL-compatibilis eszközhöz, mint pl billentyűzet, kapcsolók, fények vagy nyomtató. Dióhélyban a CPU program tud írni 0-t vagy 1-et bármely vonalra, vagy tudja olvasni bármely vonal bemeneti állapotát, nagy rugalmasságot nyújtva ezzel. Egy kis CPU központú rendszer PIO chippel gyakran pótolni tud egy teljes alaplapot tele SSI vagy MSI chippel, különösen egy beágyazott rendszerben.

3-55. ábra. Egy 8255A PIO chip.

Bár a CPU több uton is be tudja állítani a 8255A chipet, azzal, hogy a állapotregisztereket betölti a chippel eggyütt, mi a működés egyszerűbb módjaira fogunk koncentrálni. A legegyszerűbb út a 8255A használatához három független 8 bites port, ay A, B és C. A portok kisegítésére van egy 8 bites záróregiszter. A porton lévő vonalak beállításához, a CPU csak beír egy 8 bites számot a megfelelő regiszterbe, és a 8 bites szám megjelenik a vonal kimenetén és ottmarad, amíg a regiszter újraíródik. Hogy bemenetként használjon egy portot,a CPU csak olvassa a megfelelő regisztert.

Más működési módokat is biztosít a külső eszközökkel való kapcsolatra. Például, hogy kimenő jelet küldjön egy eszközre, ami nincs mindig kész az adatok fogadására. a 8255A tud adatot küldeni egy kimeneti portra és várni amíg az eszköz visszaküld egy impulzust, mondva, hogy fogadta az adatot és vár a többire. Az ilyen impulzusok fogadásához és a CPU számára elérhetővé tevéséhez szükséges irányító rész bele van építve a 8255A hárdverjébe.

A 8255A működési diagrammjából láthatjuk, hogy a 24 láb három porthoz való hozzáadásakor, van 8 vonal ami közvetlenül kapcsolódik az adatbuszhoz, egy chip kiválasztó vonal, olvasó és író vonal, két címzés vonal, és egy vonal a chip rezeteléséhez. A két címzés vonal kiválaszt egyet a 4 belső regiszterből, megfelelteti az A, B, C portoknak és az állapotregiszternek, amleynek bitjei meghatározzák melyik port van bemenetre, kimenetre és más feladatokra fenntartva. Általában a két címzés bit a címzés busz alacsony rendű bitjeihez kapcsolódik.

3.7.2 Cím Dekódolás

Emlékezzünk vissza arra, amikor szándékosan bizonytalanok voltunk abban, hogyan szerez a érvényt a chip kiválasztás a memórián és megnéztük az I/O chipeket.. Itt az, hogy tüzetesebben megvizsgáljuk, hogy is működik ez. Vizsgáljunk meg egy 16 bites beszerkesztett számítógépet, ami áll egy CPU-ból, egy 2Kx8 bájtos EPROM-ból a programhoz, egy 2Kx8 bájtos RAM-ból az adatoknak, és egy PIO-ból. Ezt a kis rendszer használható egy olcsó játék agyaként, vagy sima készülékként. Együegy termékben az EPROM helyettesíthető egy ROM-mal.

A PIO két út közül az eggyikként választható ki; létező I/O eszközként, vagy a memória egy részeként. Ha az I/O eszközként való használatot választjuk, akkor ki kell válasszuk, hogy egy meghatározott busz vonalat használjon, ami megmutatja, hogy I/O eszközként van hivatkozva rá, nem pedig memóriaként. Ha a másik

megközelítést használjuk, memória terület I/O, akkor meg kell határozzuk a memória 4 bájtjával, a három prothoz és az irányító regiszterhez. A választás valamelyest önhatalmú. Mi a memorz-mapped I/O-t választjuk, mert az megmutat néhány igen érdekes problémát, az I/O interfészében.

Az EPROM-nak 2 bájtra van szüksége a címzési helyből, a RAM-nak szintén 2 bájt kell, a PIO-nak pedig 4 bájt. Mivel apéldánkban a címzési hely 64K, ki kell válasszuk, hogy hova tegyük a három eszközt. Egy lehetséges választást mutat a 3-56. ábra. Az EPROM címeket foglal a 2K-ig, a RAM elfoglalja a címeket 32K-tól 34K-ig, és a PIO elfoglalja a címzési hely felső négy bájtját, 65532-től 65535-ig. A programozók nézőpontjából mindegy mely címeket használjuk; mégis, az interfésznél számít. Ha az I/O-n keresztüli PIO címzést választjuk, akkor nem igényel több memóriacímet (de szükség lesz négy I/O címre).

3-56. ábra. Az EPROM, a RAM és a PIO elhelyezkedése a mi 64K címzési helyünkön.

A 3-56. ábra szerinti címbeosztással minden 16 bites memóriacímet a 00000xxxxxxxxxxx (bináris) formából kellene kiválasztani. Más szavakban, bármelyik cím, aminek 5 magasszintű bitje mind 0, beleesik a memória alsó 2K-jába, ennélfogva az EPROM-ba. Így az EPROM chip kiválasztója egy 5 bitű összehasonlítóba vezethet, egyikük bemenete tarósan nullához vezet.

3-57. ábra. (a) Teljes cím dekódolás. (b) Részleges cím dekódolás.

Egy jobb út ugyanazon hatás eléréséhez, ha egy öt bemenetű OR kaput használunk az öt A11-A15 címekhez csatolt bemenettel. Ha és csak akkor ha mind az öt vonal 0, lesz a bemenet 0, következésképpen érvényesítve CS-t(ami alacsony igényelt). Sajnos nincs öt bemenetű OR kapu a szabványos SSI sorozatban. A legközelebbi, ami járható, egy nyolc bemenetű NOR kapu. Leföldelve három bemenetet és invertálva a kimenetet, mégis megkaphatjuk a helyes jelet, ahogy a 3-57(a). ábrán látszik. Az SSI chipek nagyon olcsóak, ami kivételt tesz kivételes helyzetekben, tehát nem gond, ha szakszerűtlenül használjuk. Szokásosan, a nem használt bemenetek nincsenek feltüntetve az áramkör diagramokon.

Ugyanazt az elvet használhatjuk a RAM-nál is. Azonban a Ram vissza kell jelezzen a 10000xxxxxxxxxxx formájú bináris címekre, ezért egy újabb invertálóra van szükség, ahogy az az ábrán látható. A PIO cím dekódolás kicsit bonyolultabb, mert az a négy címmel van kiválasztva az 11111111111111xx formából. Egy lehetséges áramkör ami csak akkor érvényesíti CS, amikor a helyes cím megjelenik a cím az ábrán látható buszon. Ez két nyolc bemenetű NAND kaput, hogy ellássa egy OR kapu szerepét. Hogy megépítsük a 3-57(a). ábrán látható cím dekódóló logokai áramkört SSI-t használva szükség van hat chipre - négy nyolc bemenetű chip, egy OR kapu és egy chip három invertálóval.

Azonban, ha a számítógép tényleg csak egy CPU-ból, két memória chipből és a PIO-ból áll, használhatunk egy trükköt, hogy nagymértékben leegyszerűsítsük a cím dekódolást. A trükk azon alapszik, ahogy minden EPROM címez, és ahogy csak EPROM címez, 0 van a magas szintű biten, az A15. Ezért közvetlenül átvezethetjük CS-t A15-be, ahogy a 3-57(b) ábrán látható.

Ennél a pontnál az elhatározás, hogy a RAM-ot 8000H-ba tesszük, sokkal kevesebbnek tűnhet, mint önhatalmúságnak. A RAM dekódolást semmi nem tudja

megcsinálni úgy, hogy a lehetséges címek, csak a 10xxxxxxxxxxxxxx formából legyenek a RAM-ba, így 2 bit elég a dekódoláshoz. Hasonlóan, bármely cím, ami 11-el kezdődik, PIO cím kell legyen. A teljes dekódoló logikai áramkör most két NAND kapu és egy inverter. Mivel egy invertert meg lehet csinálni egy NAND kapuból úgy, hogy csak simán összekötjük a két bemenetet, egy sima négyes NAND chip már több mint elég.

A 3-57(b). ábra cím dekódoló logikai áramkörét részleges cím dekódolónak nevezzük, mert nincs használva a teljes cím. Ennek meg van az a tulajdonsága, hogy egy olvasás a 0001000000000000, 0001100000000000 vagy 0010000000000000 címekről, ugyanazt az eredményt fogja adni. Valójában, minden cím a címtartomány alsó feléből az EPROM-ot fogja kiválasztani. Mivel a különleges címek nincsenek használva, nem lehet kárt okozni, de ha valaki tervez egy számítógépet, ami sűrűn előfordulhat a jövőben (egy valószínűtlen előfordulás egy játékban), a részleges dekódolást el kellene kerülni, mert túl nagy címtartományt köt le.

Egy másik általános cím dekódolási teknika, ha olyan dekódert használunk, amilyen a 3-13. ábrán látható. A három bemenet hozzákapcsolva a három magasszintű cím vonalhoz, nyolc kimenetet kapunk, megfelelően az első 8K, második 8K, stb. címeknek. Egy számítógépben nyolc RAM-mal, mind 8Kx8-as, egy olyan chip teljes dekódolást ad. Egy számítógép nyolc 2Kx8-as memóriachippel, egy sima dekóderrel szintén elég, és azt nyújtja, hogy a memóriachipek mind meg vannak határozva a címtartomány könnyen érthető 8K-s nagy darabjaiban. (Emlékezzen a korábbi megjegyzésünkre, hogy a memória helyzete és az I/O chipek a címtarományhoz képest számítanak.)

***[198-201]

Fordította: Punyi Róbert
h938409

198-201

3.8 **ÖSSZEFOGLALÁS**

A számítógépeket integrált áramkörös chipekkel készítik. Ezekben a chipekben apró kapcsolók vannak, amiket kapuknak hívunk. A legismertebb kapuk az AND (és), OR (vagy), NAND (nem, és), NOR (nem, vagy) és a NOT (nem). Az egyszerű áramköröket közvetlenül működő, individuális kapukkal lehet szerelni.

A bonyolultabb áramkörök a multiplexerek, demultiplexerek, kódolók, dekódolók, kapcsolók és ALU-k. Tetszőleges Boolean funkciókat PLA-val lehet programozni. Ha sok Boolean funkcióra van szükség, a PLA-k még hatékonyabbak. A Boolean algebra szabályait felhasználva egy áramkört egy másik áramkörré lehet átalakítani. Egyes esetekben így gazdaságosabb áramkörök készíthetők.

A számítógépes aritmetikát az összeadók végzik. Egy 1 bites teljes összeadó két fél összeadóból áll össze. Egy multibites szó összeadása úgy jön létre, hogy összetett teljes összeadókat kapcsolunk össze úgy, hogy mindegyik kimenete a bal szomszédja felé legyen.

A statikus memóriák összetevői a tárolók és a flip-flopok, amelyek mindegyike 1 bit információ tárolására képes. Ez kombinálható lineárisan az oktális tárolókba és a flip-flopokba vagy logaritmikusan a teljes méretű szó orientált memóriába. A memóriák típusai a RAM, ROM, PROM, EPROM, EEPROM, és flash. A statikus RAM-okat nem kell frissíteni. Az adatokat addig tárolják, amíg áram alatt vannak. A dinamikus RAM-okat ugyanakkor állandóan frissíteni kell, hogy kompenzáljuk a chip kis befogadóképességét.

A számítógépes rendszerek összetevői buszokkal kapcsolódnak egymáshoz. Egy átlagos CPU legtöbb lába egy-egy buszvonalat vezérel. A buszvonalakat cím, adat és kontrollvonalakra oszthatjuk. A szinkron buszokat egy órajel generátor vezérli. Az aszinkron buszok összehangolják a szolgát és a mestert.

A Pentium II a modern CPU egyik példája. A rendszerek, amelyek ezt használják rendelkeznek egy memória busszal, egy PCI busszal, egy ISA busszal és egy USB busszal. A PCI busz egyszerre 64 bit információt tud szállítani 66 Mhz-el, ami elég gyors a perifériáknak, de nem elég gyors a memóriáknak.

A kapcsolók, lámpák, a nyomtatók és más I/O eszközök párhuzamos I/O chipeket használnak, mint a 8255A. Ezeket a chipeket konfigurálhatjuk, hogy az I/O hely és a memóriahely részei legyenek. Teljesen vagy részben dekódolhatók az alkalmazástól függően.

FELADATOK

1. Egy logikatanár betér egy autós étterembe és azt mondja: Egy hamburgert vagy egy hot dogot és sült burgonyát kérek. Sajnos a szakács éppenhogy átcsúszott matematikából az iskolában és nem tudja, hogy az "és" fölényben van-e a "vaggyal" szemben. Ami őt illeti, az egyik olyan, mint a másik. A következő esetek közül melyik a helyes? (Az angol "vagy" kizárólagos "vagyot" jelent.)

a. Csak egy hamburgert
b. Csak egy hot-dogot
c. Csak sült burgonyát
d. Hot dogot és sült burgonyát

e. Hamburgert és sült burgonyát
f. Hot dogot és hamburgert
g. Mind a hármat
h. Semmit

2. Egy misszionárius eltéved Dél-Kaliforniában, és egy Y útkereszteződésnél áll. Tudja, hogy két motoros banda uralja a területet. Az egyik mindig igazat mond, a másik mindig hazudik. Azt akarja tudni, hogy melyik út vezet Disneylandbe. Mit kérdezzen?

3. Egy változónak négy Boolean funkciója van és két változónak 16 funkciója. Három változónak hány funkciója van? És n változónak?

4. Használjon igazságtáblázatot és mutassa meg, hogy P = (P és Q) vagy (P és nem Q)!

5. Mutassa meg, hogy hogyan lehet elkészíteni az AND (és) funkciót két NAND (nem, és) kapuból!

6. Találja meg $A\overline{B}$ komplementerét! Használja DeMorgan törvényét!

7. A 3-12 ábra háromváltozós multiplexer chipjét használva írjon le egy működést, amelynek kimenete egyenlő a bemenetével, azaz a kimenet akkor és csakis akkor 1, ha a bemenet egyik páratlan száma 1!

8. Tegye fel a gondolkodó sapkáját! A 3-12 ábra háromváltozós multiplexer chipje képes programozni a négy Boolean változó egyik tetszőleges funkcióját. Írja le hogyan, és rajzoljon egy logikai diagramot a 0 függvényhez, ha az angol szóban páros számú betű van. (pl.: 0000 = nulla = 4 betű -> 0; 0111 = hét = 5 betű -> 1;1101 = tizenhárom = 8 betű -> 0). Tanács: Ha a negyedik bemeneti változót D-vel jelöljük, a nyolc bemeneti vonalat V_{CC} -hez kapcsolhatjuk. Jelölés: D, vagy D.

9. Rajzolja meg egy 2 bites demultiplexer logikai diagramját, egy áramkört, amelyiknek az egyes bemeneti vonala a négy kimeneti vonal egyikéhez van irányítva, a két ellenőrző vonal állapotától függően!

10. Rajzolja meg egy 2 bites kódoló logikai diagramját: 4 bemeneti vonallal, amelyek egyikén mindig folyik az áram, és a két 2 bites bináris kimeneti vonal értéke megmondja, hogy melyik input van áram alatt!

11. Rajzolja újra a 3-15 ábra PLA-ját elég részletesen ahhoz, hogy a 3-3 ábra logikai funkcióit meg lehessen mutatni! Mindenképp mutassa meg, hogy mik a kapcsolatok a két mátrix között!

12. Mit csinál ez az áramkör?

13. Egy MSI chip egy 4 bites adó. Négy ilyen chipet úgy is össze lehet kötni, hogy egy 16 bites adó legyen. Mennyi lábának kell lenni egy 4 bites összeadó chipnek? Miért?

14. Egy n bites összeadó megkonstruálható n teljes összeadók sorba kapcsolásával, az átvitel állapotban, C_i , az i-1 állapot kimenetéből származik. Az átvitel 0 állapotban, C_0, 0. Ha minden állapot T nsec-ot igényel az összeg és átvitel előállításához, az átvitel i állapotban nem lesz valós, amíg iT nsec el nem telik az összeadás kezdetétől. Nagy n-re az idő ahhoz, hogy az átvitel beálljon a magasabb rendű állapotba nem kívánatosan hosszú lehet. Tervezzen egy összeadót, ami gyorsabban dolgozik! Tanács: minden C_i kifejezhető A_{i-1} és B_{i-1}

operandus bitek szempontjából úgy, mint C_{i-1}. Ennek a relációnak a használatával lehetséges kifejezni C_{i-t} , egy input függvényeként a 0 állapotokról i-1-re, így az összes átvitel előállítható egyidejűleg.

15. Ha a 3-19 ábra összes kapuja késleltetve van 10 nsec-cel, és az összes többi késleltetést mellőztük, akkor mikor lesz az áramkörnek érvényes áramköri bitje?

16. A 3-20 ábra ALU-ja alkalmas 8 bites teljes összeadás elvégzésére. Alkalmas-e teljes kivonás elvégzésére? Ha igen, magyarázza meg, hogyan! Ha nem, alakítsa át, hogy alkalmas legyen!

17. Néha hasznos egy 8 bites ALU-nak, (mint a 3-20 ábrán) hogy tudjon készíteni konstans 1-et, mint kimenetet. Adjon két módszert, hogy ez lehetséges legyen. Mindegyik módnál adja meg a 6 ellenőrző jelzés értékeit.

18. Egy 16 bites ALU 16 darab 1 bites ALU-ból áll, amelyek mindegyikének 10 nsec ideje van. Ha van egy 1 nsec-es késleltetés egyik ALU-tól a következőhöz, mennyi ideig tart, amíg megjelenik a 16 bites összeadás eredménye?

19. Mi az S és R bemenetek nyugalmi állapota egy SR tárolóhoz, amely két NAND (nem, és) kapuból tevődik össze?

20. A 3-26 ábra áramköre egy flip-flop, amely az óra felfelé tartó ágán van. Módosítsa ezt az áramkört, hogy egy olyan flip-flophoz jusson, amely az óra leszálló ágán van!

21. Segítsen egyeztetni a részleteket az új személyi számítógépében, vegye fel a kapcsolatot a kezdő SSI chip gyártókkal. A kliensei közül egy arra gondol, hogy kikapcsol egy négy D flip-flop-ot tartalmazó chipet, mind a Q-t, mind a Q-t, egy potencionálisan fontos vásárló kérésére. A javasolt tervezésnek mind a négy óra jele sorba van kapcsolva, szintén kívánságra. Nincs beprogramozva se üresre állítva. Az ön feladata megadni a tervezésnek egy professzinális kiszámítását.

22. A 3-29 ábra 4*3-as memóriájának 22 ÉS kapuja és 3 OR kapuja van. Ha az áramkőr 256*8-ra lenne kibővítve hány ÉS és VAGY kapu lenne?

23. Mivel több és több memóriát integrálnak egy chipre, a címlábak száma is növekszik. Azonban gyakran kellemetlen a nagy számú címlábak a chipen. Hogy lehet 2^n memóriát megcímezni kevesebb, mint n láb használatával?

24. Egy számítógép 32 bites adatbusszal 1M*1 dinamikus RAM memória chipet használ. Mennyi a legkevesebb memória (byte-okban), amit a gép használhat?

25. Az időzítés diagramra hivatkozva (3-37 ábra) tegyük fel , hogy az órát 25 nsec helyett 40 nsec időtartamra lassítjuk le, de az időzítésnek változatlannak kell maradnia. Legrosszabb esetben mennyi időbe telne a memóriának, hogy T_3 idő alatt a buszra tegye az adatot, miután MREQ érvénybe lép?

26. Megint a 3-37 ábrára hivatkozva tegyük fel, hogy az óra 40 Mhz-en van, de a T_{AD} 16 nsec-re van kibővítve. Használhatunk-e 40 nsec-es memória chipet?

27. A 3-37 (b) ábrán a T_{ML} legalább 6 nsec. Van-e olyan chip, amelyben ez negatív? Más szavakkal: Tudja-e a CPU-t érvényesíteni MREQ-t mielőtt a cím biztos? Miért, vagy miért

nem?

28. Tegyük fel, hogy a 3-41-es ábra átalakítása a 3-37-es ábra buszán történik. Mennyivel nagyobb sávszélesség szükséges az egyedüli átvitelek esetében? Most tegyük fel, hogy a busz 32 bites. Válaszoljon ismét a kérdésre!

29. Mutassa meg a 3-38 -as ábra címeinek átviteli idejét, mint T_{A1} és T_{A2}, és $\overline{MREQ}$ átviteli idejét, mint T_{MREQ1} és T_{MREQ2} és így tovább! Írjon le minden egyenlőtlenséget!

30. A legtöbb 32 bites busz engedélyezi a 16 bites írás-olvasást. Van valami kétsége, hogy hová tegye az adatot? Beszélje meg!

31. Több CPU-nak speciális busz ciklus típusa van a megszakítás fogadáshoz. Miért?

32. Egy 10MHz-es PC / AT-nek négy áramkörre van szüksége, hogy elolvasson egy szót. Mekkora busz sávszélességet használ fel a CPU?

33. Egy 32 bites CPU A2-A31címekkel, minden memóriacímet elfoglal. A szavak 4 byte-tal vannak címezve, és a fél szavakat páros byte-okhoz kell címezni. A byte-ok bárhol lehetnek. Hány szabályos kombináció van a memóriaolvasáshoz, és hány láb kell hozzájuk? Adjon két választ és mindegyikhez egy esetet.

34. Miért nem lehet a Pentium II 32 bites PCI busszal dolgozni anélkül, hogy vesztene a funkcionalitásából?
Más számítógépek 64 bites adatbusszal tudnak dolgozni 32 bites, 16 bites és 8 bites adatátvitellel.

35. A CPU-nak 1 és 2 szintű cache-ja (gyorsítótára) van, 5 és 10 nsec-es elérési idővel. A fő memória elérési ideje 50 nsec. Mi az átlagos elérési idő, ha 20% 1 szintű és 60% 2. szintű.

36. Valószínűnek tartja-e, hogy egy kis picoJava II beágyazott rendszer egy 8255A chipet használ?

***[202-205] javított!!

4

A mikroarchitektura szintje

A digitális logika szintje felett a mikroarchitektura szintje van. Ennek a feladata végrehajtania a felette lévő ISA (Insturction Set Architecture / Utasításkészlet Architektura) szintet, ahogy ezt az 1-2-es ábra illusztrálja. A mikroarchitektura szintjének konstrukciója éppen annyira függ az ISA-tól, mint a számítógép árától és teljesítményétől. Számos modern ISA-nak, különösen a RISC konstrukcióknak, vannak egyszerű utasításai, amik általában végrehajthatók egy óraciklusban. Az összetettebb ISA-k, mint a Pentium II, esetleg számos ciklust igényelnek egyetlen utasítás végrehajtásához. Egy utasítás végrehajtása az operandusok memóriabeli elhelyezkedését, azok olvasását és az eredmények visszamentését igényelheti a memóriába. A műveletek ütemezése egy egyszerű utasításon belül gyakran vezet különböző úton a vezérléshez, mint az egyszerű ISA-knál.

4.1 Egy példa a mikroarchitekturára

Ideálisan, a mikroarchitektura tervezése általános alapjainak megmagyarázásával szeretnénk bemutatni ezt a témát. Sajnos nincsenek általános alapok, mindegyik speciális eset. Következésképpen, ehelyett meg fogunk vitatni egy részletes példát. A példa ISA-nkhoz a Java virtuális gép egy részhalmazát választottuk, ahogy az 1-es fejezetben ígértük. Ez a részhalmaz csak egész utasításokat tartalmaz, így **IJVM**-nek neveztük el. Vitatni fogjuk az egész JVM-et az 5. fejezetben.

A mikroarchitektura leírásánál felülről indulunk ki, ahol végrehajtjuk az IJVM-et. Az IJVM-nek néhány viszonylag összetett utasítása van. Sok hasonló architektura sokszor a mikroprogramozáson keresztül lett megvalósítva, ahogy ezt az 1. fejezetben tárgyaltuk. Habár az IJVM kicsi, jó kiindulási pont az utasítások vezérlése és szekvenciálása leírásához.

A mikroarchitekturánk tartalmazni fog egy mikroprogramot (a ROM-ban), aminek feladata IJVM utasításokat betölteni, dekódolni és futtatni. Mi nem tudjuk használni a Sun JVM fordítót a mikroprogramhoz, mert szükségünk van egy apró mikroprogramra, ami hatékonyan vezérli az egyes kapukat az aktuális hardverben. Ellentétben, a Sun JVM fordító C-ben volt írva a mozgathatóság miatt, és nem tudja olyan részletességgel kezelni a hardvert, mint amire nekünk szükségünk van. Mivel az aktuális hardver csak a 3. témában leírt alapkomponensekből használ elemeket, elméletben, e téma teljes megértése után az olvasónak képesnek kellene lennie elmenni és venni egy nagy táskányi tranzisztort és megépíteni a JVM gép ezen részét. Azok a tanulok, akik sikeresen teljesítik ezt a feladatot, extra kreditet kapnak (és egy teljes pszichiátriai vizsgálatot).

A mikroarchitektura vázlatához megfelelő modell az, hogy úgy gondoljunk a konstrukcióra, mint egy programozási problémára, ahol minden utasítás az ISA szintjén egy függvény, amit egy főprogram meghív. Ebben a modellben a főprogram egy egyszerű, vég nélküli ciklus, ami meghatározza a segítségül hívandó függvényt, meghívja a függvényt, majd az egészet újrakezdi, nagyon hasonlóan a 2-3-as ábrához.

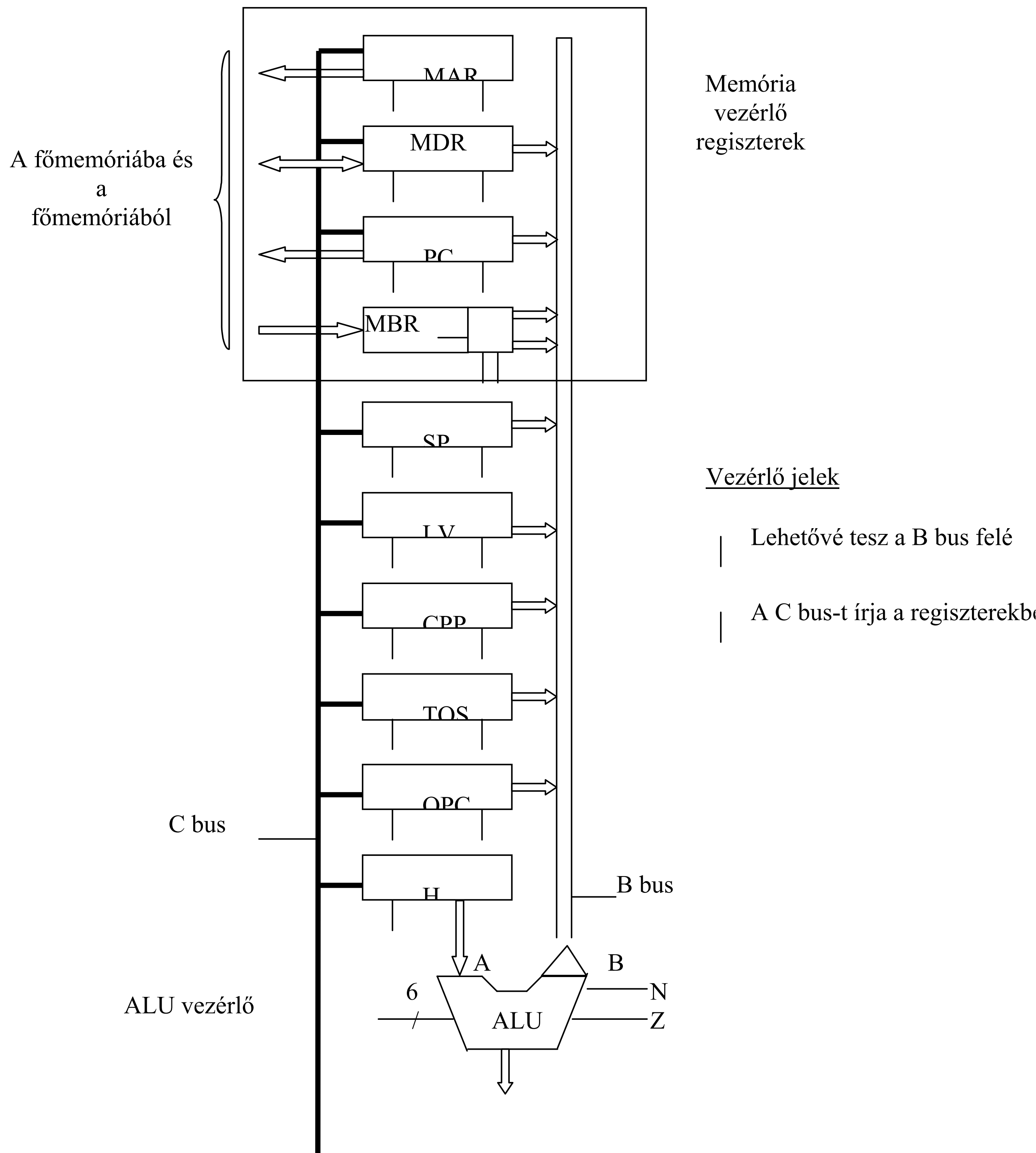
A mikroprogramnak van egy változó halmaza, amit a gép állapotának (**state**-jének) neveznek, ami minden függvény számára hozzáférhető. Az állapot (state) képzése közben minden függvény legalább néhányat megváltoztat a változókból. Például a Program Counter (PC) az állapot (state) része. Ez jelöli a következő végrehajtandó függvényt (úgymint az ISA utasítást) tartalmazó memóriahelyet. Minden egyes utasítás végrehajtása alatt a PC úgy módosul, hogy a következő végrehajtandó utasításra mutasson.

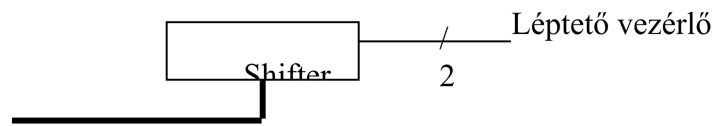
Az IJVM utasítások rövidek és aranyosak. Mindegyik utasításnak van néhány mezője, általában egy vagy kettő, amelyek mindegyikének van valami speciális jelentése. Minden utasítás első mezője az **opcode** (rövidítése az **operációs kód**nak), ami azonosítja az utasítást, megmondja, hogy ez egy ADD, vagy egy BRANCH, vagy valami más. Számos utasításnak van egy további mezője, ami az operanduszt határozza meg. Például az utasítások, amelyek hozzáférnek egy helyi változóhoz, igényelnek egy mezőt, ami megmondja melyik változó.

A végrehajtás ezen modellje (néha fetch-executed ciklusnak nevezik), hasznos az összefoglalásban és esetleg a végrehajtás alapja is lehet azon ISA-k számára, mint az IJVM, aminek összetett utasításai vannak. Az alábbiakban le fogjuk írni, hogy működik ez, hogyan néz ki a mikroarchitektura, és hogyan vezérlik ezt a mikroutasítások, amelyek mindegyike vezérli az adatutat 1 ciklusig. A mikroutasítások listája együtt alkotja a mikroprogramot, amit részletesen fogunk bemutatni és tárgyalni.

4.1.1 The Data Path (Az adat út)

A **data path** a CPU azon része, amely az ALU-t tartalmazza, annak bemeneteit és kimeneteit. A példa mikroarchitekturánk adat útját a 4-1-es ábra mutatja. Amíg ez gondosan optimizálva lett IJVM programok fordítására, ez közel hasonló a legtöbb gépben használt adat úthoz. Ez 32 bit-es regiszterek egy egész sorát tartalmazza, amihez szimbolikus neveket rendeltünk, mint a PC, SP, és az MDR. Habár ezek a nevek közül néhány általánosan ismert, fontos megérteni, hogy ezek a regiszterek csak a mikroarchitektura szintjén elérhetőek (a mikroprogram számára). Azért kapják ezeket a neveket, mert általában felvesznek egy értéket az ISA szint architekturában lévő ugyanazon nevű változatának megfelelően. A legtöbb regiszter képes….





4.1 Ebben a témában használt példa architektura adat útja

***[206-209]

Számítógép architektúra
Fordítás a 206-209. oldalig

… tartalmát a B buszra irányítani. Az ALU kimenete vezérli a léptető áramkört és a C buszt, aminek az értékei egyszerre egy, vagy akár több regiszterbe is beírhatók. Jelenleg még hiányzik az A busz; később azt is hozzávesszük.
Az ALU azonos a 3-19. és a 3-20. ábrákon bemutatottal. Funkcióját hat vezérlőjel határozza meg. A 4-1. ábra 6-os jelzésű rövid, átlós vonala utal ezekre. Az F0 és az F1 az ALU műveletet határozzák meg, az ENA és az ENB egyénileg működtetik a bemeneteket, az INVA a bal bemenet értékét negálja, az INC pedig, lényegében egyes átvitelt előállítva a legkisebb helyiértékű bithez egyet ad. Az összesen 64 lehetséges kombináció közül azonban nem mindegyik használható érdemlegesen.
A 4-2. ábra néhány fontosabb variációt mutat be. Ezen funkciók közül IJVM-hez nem feltétlenül szükséges mindegyik, azonban a teljes JVM-hez jól jöhet jónéhány. Sok esetben több lehetőség is ugyanazt fogja eredményezni. A táblázatban a “+” aritmetikai összeadást, a “-“ aritmetikai kivonást jelent, így például a –A az A kettes komplemensét jelöli.
(4-2. ábra: Hasznos ALU jelkombinációk és működésük)
A 4-1. ábra ALU-ja két adatbeviteli ágat igényel: a bal bemenetet (A), és a jobb bemenetet (B). A bal bemenethez a H jelű regiszter kapcsolódik. A jobb bemenethez kapcsolódik a B busz, mely betölthető a kilenc forrásregiszter bármelyikéből (ezeket a buszra mutató kilenc szürke nyíl jelzi). A dolognak egy másik megvalósításával, mely két teljes busszal operál, és előbbiektől különböző adatátviteli beállításokkal bír, még később foglalkozunk ebben a fejezetben.
A H-t elsősorban úgy lehet betölteni, hogy azt az ALU funkciót választjuk, mely éppen a jobb bemeneten keresztül vezet a B buszról az ALU kimenetéhez. Egy példa erre, ha az ALU bemenetek értéke 1, s egyedül az ENA szerepel negálva, tehát a bal bemenet 0 lesz. A B buszon szereplő értékhez 0-t hozzáadva éppen a B-n lévő értéket kapjuk. Ez az eredmény aztán módosítás nélkül haladhat át a léptetőn, és elraktározódik a H regiszterben.
A fenti függvényeken kívül két másik lehetséges vezérlőjel is használható, melyek az előbbiektől függetlenül vezérlik az ALU kimenetét. Az SLL8 (Shift Left Logical) 1 bájttal balra lépteti a kimenet tartalmát, így 0-val töltve meg a 8 legalacsonyabb helyiértékű bitet. Az SRA1 (Shift Right Arithmetic) egy bittel jobbra tolja tartalmát, változatlanul hagyva a legmagasabb helyiértékű bitet.
Világos, hogy egy cikluson belül is lehetséges egyazon regisztert olvasni, illetve oda írni. Lehet például SP-t tenni a B buszra, kikapcsolni az ALU bal bemenetét, bekapcsolni az INC jelet, és tárolni az eredményt SP-ben, így növelve az SP-t eggyel. Hogyan olvasható és írható egy regiszter egyetlen cikluson belül téves eredmény nélkül? A megoldás az, hogy az írás és az olvasás a cikluson belül valójában nem ugyanabban az időben történik. Ha egy regisztert az ALU jobb bemeneteként adunk meg, az értékét már a ciklus elején a B buszra irányítjuk, és folyamatosan ott tartjuk az egész ciklus ideje alatt. Az ALU aztán teszi a dolgát, elkészíti az eredményt, mely a léptetőn keresztül a C buszra távozik. A ciklus vége felé, amikor az ALU és a léptető kimenetei stabilak, egy órajel egy vagy több regiszterbe irányítja a C busz tartalmát. Ezen regiszterek egyike esetleg ugyanaz lehet, amely a B buszt ellátta adattal. Az adat útjának precíz időzítése teszi lehetővé az egy cikluson belül egy

regiszterben lezajló olvasás-írást; mint ahogy azt az alábbiakban leírjuk.

AZ ADATÚT IDŐZÍTÉSE

Ezen események időzítését a 4-3. ábra mutatja. Itt minden ciklus kezdetét egy rövid impulzus jelzi, mely a főórából származik (3-21c. ábra). Az impulzus leeső élén állítódnak be a kapukat vezérlő bitek. Ennek ideje (?w) véges, és pontosan megállapítható. Aztán a B buszhoz szükséges kijelölt regiszter az adatokat a B buszra küldi. ?x időbe telik, mire ez az érték stabillá válik, majd az ALU és a léptető megkezdi működését a megadott adatokkal. Újabb x idő múlva az ALU, és a léptető kimenetei stabillá válnak. Egy szükségszerű ?z idő alatt az eredmények a C buszon keresztül betöltődnek a regiszterekbe, ahonnan a következő impulzus emelkedő élén olvashatók lesznek. Az onnan való betöltés olyan gyorsasággal mehet végbe, hogy ha néhány bemeneti regiszter értékét megváltoztatjuk, a hatások a C buszon jóval a regiszterekből történő olvasás után érezhetők csak. Még az impulzus emelkedő élén leáll a B buszt vezérlő regiszter, így készül fel a következő ciklusra. Az MPC-t, az MIR-t és a memóriát az ábrán láthatjuk; szerepüket rövidesen tárgyaljuk.

Fontos észrevenni azt, hogy bár az adatúton nincsenek tároló elemek, a terjedési idő mégis pontosan meghatározható rajta. Ha a B busz értékét megváltoztatjuk, a C buszon ez addig nem okoz változást, míg egy meghatározott idő el nem telik (ez a lépésenkénti késések miatt van). Következésképpen, még ha egy tár meg is változtatja a bemeneti regiszterek egyikének tartalmát, az adat már jóval előbb biztonságban van a regiszterben, minthogy a (most helytelen) érték a B buszra (vagy a H-ba) kerülve elérné az ALU-t.

Ahhoz, hogy ez a szerkezet működjön is, szigorú időzítés szükséges, továbbá hosszú ciklusidő, minimális, pontos átfutási idő az ALU-n, és hogy az adatok gyorsan töltődjenek át a regiszterekbe a C buszról. Ezek ellenére gondos mérnöki munkával az adatutat meg lehet tervezni úgy, hogy pontosan működjön.

Az adatút-ciklust értelemszerű alciklusokra felbontva is tekinthetjük. Az első alciklus megkezdését az impulzus elcsendesedése váltja ki. Az alciklusok tevékenységét az eltelt idővel együtt (zárójelben) az alábbiakban mutatjuk meg.

1. A vezérlő jelek beállítása (?w)
2. A regiszterek tartalma betöltődik a B buszra (?x)
3. Az ALU és a léptető működése (?y)
4. Az eredmények a regiszterekbe töltődnek a C buszon keresztül (?z)

A következő órajel emelkedő élén az eredmények a regiszterekben tárolódnak.

Azt mondtuk, a legjobb, ha az alciklusokat közvetettnek tekintjük. Ezalatt azt értjük, hogy nincs órajel, vagy más határozott jelzés, mely megmondja az ALU-nak, mikor dolgozza fel az adatokat, vagy küldje el az eredményeket a C buszra. Valójában az ALU és a léptető állandóan működnek. Mindazonáltal a ?w+?x idő alatt a bevitt adatok használhatatlanok. Ugyanígy a ?w??x??y idő alatt az eredmények is rosszak. Az határozott, az adatutat vezérlő jelzések csupán az órajel leszálló ága, melynek hatására a B busz töltésével elindul az adatút, továbbá az órajel felszálló ága, amikor a C busz adatai a regiszterekbe töltődnek. Az alciklusok határait értelemszerűen az érintett áramkörök működési ideje határozza meg. A tervezőmérnökökön múlik, hogy gondoskodjanak róla, a ?w??x+?y+?z idő megfelelően a következő impulzus erősödése előtt véget érjen, a regiszterek folyamatos működését biztosítandó.

MEMÓRIA MŰVELETEK

Gépünk két különböző módon kommunikál a memóriával: egy 32 bites, szócímzésű,

és egy 8 bites, bájtcímzésű memóriakapun keresztül. A 32 bites kaput két regiszter vezérli, a 4-1. ábrán bemutatott MAR (Memory Address Register, memória címregiszter) és MDR (Memory Data Register, memória adatregiszter). A 8 bites kaput egyetlen regiszter vezérli, a PC, mely 1 bájtot olvas be az MBR alacsony helyiértékű 8 bitjének helyére. Ez a kapu csak olvasni tud a memóriából, oda írni nem.
Az említett regiszterek mindegyikét (és a 4-1. ábra összes regiszterét) egy vagy két vezérlő jel irányítja. A regiszter alatti üres nyíl azt a vezérlőjelet adja, mely bekapcsolja a regiszter kimenetét a B buszra. Mivel a MAR nincs összekötve a B busszal, nem kaphat bekapcsolásra utasító jelet. A H sem tud, mert mindig be van kapcsolva, lévén az egyetlen lehetséges bal ALU bemenet.
A regiszter alatti vastag fekete nyíl jelzi azt a vezérlőjelet, melynek hatására a C buszról a regiszterbe töltődik az adat. Mivel az MBR nem tölthető be a C buszról, nincs is írást vezérlő jelzése (bár két másik bekapcsoló jele van, mint az alatta látható). A memória olvasásához vagy írásához elengedhetetlen, hogy a megfelelő memóriaregiszter be legyen töltve, aztán kell adni egy olvasás vagy írás jelzést a memóriának (ez nincs feltüntetve a 4-1. ábrán).
A MAR szócímeket tartalmaz úgy, hogy a 0,1,2… értékeknek szavakat feleltet meg. A PC bájt alapú címzéseket tartalmaz, és a 0,1,2… értékeknek bájtokat feleltet meg. Így, ha 2-t írunk a PC-be, és memóriaolvasást indítunk, az a 2-es bájtot olvassa ki a memóriából, és az MBR alacsony helyiértékű 8 bitjébe tárolja. Ha viszont a MAR-ral tesszük ugyanezt, az a 8-11-es bájtot fogja olvasni (2-es szó), és az MDR-be tárolja azt.
Ez a funkcióbeli különbözőség szükséges, mivel a MAR-t és a PC-t a memória két különböző területének címzésére használjuk. Később ez világossá válik. Pillanatnyilag elegendő az is, hogy a MAR/MDR kombinációt ISA-szintű szavak olvasására és írására használjuk, a PC/MBR változat pedig végrehajtható ISA-szintű program olvasását szolgálja, mely program bájt adatokból áll. Minden más regiszter, mely címeket tartalmaz, a szó alapú címzést használja, mint a MAR.

Ginter Gábor
közg. prog. mat. I. évf.
h938689@stud.u-szeged.hu

***[210-213]
Nem érkezett meg. Kozák Rolland: h735077

***[214-217] javított!

214.o.
...cím sorrend (kivéve az elágazások);a mikroutasítások nem.A 2-3.ábrán jelölt véges program számláló növelése azt fejezi ki,a jelenleg végrehajtás alatt álló utasítást általában azt hajtjuk végre,amely a memóriában követi. A mikroprogramok ennél nagyobb rugalmasságot követelnek(mivel a mikroprogram sorozatoknak rövidnek kell lenniük),ezért általában nem birtokolják ezt a sajátságot.
Ehelyett mindegyik mikroutasítás félreérthetetlenül definiálja az utódját.
215.o.
Mivel a mikroprogramtár működési szempontból egy (csak olvasható) memória ezért szüksége van saját címregiszterre és saját memória-adat regiszterre.Viszont nem szükségesek neki író és olvasó jelek,mert folyamatosan olvassák.Innentől fogva a mikroprogram tár címregiszterét MPC-nek(Microprogram Counter=Mikroprogram Számláló)
fogjuk hívni.Ez a név elég ironikus,mivel a benne levő elhelyezések kétséget nem ismerően rendszertelenek,tehát a megállapodás a számlálás elnevezésre nem kézenfekvő(de kik vagyunk mi,hogy a tradíciókkal vitázzunk?).A memória-adat regisztert MIR-nek(Microinstuction Register=Mikroinstrukciós Regiszter) hivjuk.A funkciója,hogy tárolja a folyamatban levő mikroutasításokat,melyek bit-jei vezérlik az irányító jeleket ,amik az adat útat működtetik.A MIR regiszter a 4-6.ábrán ugyanazt a hat csoportot tartalmazza,mint a 4-5. ábrán.Az ADDR és J (JAM) csoportok irányítják a soron következő mikroutasítások kiválasztását és rövidesen szó lesz róluk.Az ALU csoport tartalmazza a 8 bitet,ami kiválasztja az ALU funkciókat és vezérli a léptetőt.A C bitek definiálják azt,hogy mely regiszterekbe töltsünk a C bus-ról.Az M bit-ek pedig a memória műveleteket irányítják.Végül az utolsó 4 bit a dekódert vezérli,ami eldönti, hogy mi megy a B bus-ra.Ebben az esetben egy 4-16-ig szabványos dekódert választottunk,noha a megkövetelt lehetőségek száma csak 9.Egy finomabban összehangolt tervezésben egy 4-9-ig dekóder lenne használható.
A kérdés itt az,hogy melyik éri meg jobban?Egy szabványos áramkör használata az általános áramkörökből vagy egy dirrekt ehhez rendelt?A szabványos, egyszerűbb és valószínűtlenebb,hogy baj lenne vele.Az ehhez tervezett saját kevesebb lapka területet követel,de több idő megtervezni és lehet,hogy rosszul működne.A most következőkben a 4-6. ábra leírását adjuk meg.Minden időtartamciklus kezdeténél(az időtartamnak eső szélénél az a 4-3.ábrán) a MIR feltöltődik azzal a szóval,amire az MPC mutat rá a mikroprogram tár-ban.A MIR töltési
idejét (delta)w-vel jelöljük.Ha valaki alciklusokban gondolkozik,akkor mondhatjuk ,hogy a MIR az első alatt töltődik fel.

Ha egyszer egy mikroutasítás betöltődött a MIR-be,a különböző jelek átterjednek az adat útra.Egy regiszter kiíródik a B bus-ra ,az ALU tudja melyik műveletet kell elvégezni,ezért rengeteg aktivitás észlelhető.Ez a második alciklus.A ciklus kezdeténél (delta)w+(delta)x intervallum után az ALU bemenetek stabilak.

Még egy (delta)y-nal később minden végrehajtódik és az ALU,N,Z és a léptető kimenetek stabilak.Az N és Z értékek ekkor elmentődnek egy 1-bites flip-flop párban.Ezek a bitek,mint az összes regiszter,mely a C bus-ról és a memóriából töltődik fel,az időtartamnak a növekvő sarkánál állítódnak be,közel az

adat út ciklus végéhez.Az ALU kimenet nincs tárolva csak betáplálva
a léptetőkbe.Az ALU és a léptető aktivitás a 3. alciklusban történik.
Egy újabb (delta)z intervallum után a léptető kimenet eléri a regisztereket a C buszon keresztül.Ekkor a regiszterek feltölthetők a ciklus végének a közelében (a növekvő szélénél az időtartampulzusnak a 4-3. ábrán).A 4. alciklus a regiszterek és az N és Z flip-flop-ok töltéséből áll.Ez egy kicsivel
az időtartam növekvő éle után fejeződik be,amikor az összes eredményt elmentették,az
előző memória műveletek elérhetők és az MPC -t feltöltötték.Ez a folyamat folytatódik
és folytatódik míg valaki meg nem unja és kikapcsolja a gépet.
216.o.
Az adat út vezérlésével párhuzamosan,a mikroprogramnak el kell döntenie,hogy melyik mikroutasítást hajtsa végre következőnek,mivel nem olyan sorrendben futnak,ahogy elhelyezték őket a mikroprogram táron.A következő mikroutasítás címénekkiszámítása akkor kezdődik,miután a MIR feltöltődik és stabil.Először a 9 bit-es NEXT_ADDRESS(következő cím) mezőt lemásolják az MPC
-be.Amíg ez a másolat a helyére kerül,addig a JAM mezőt megszemlélik.Ha az értéke 000,akkor semmi más sem történik;amikor a NEXT_ADDRESS másolása bevégződik ,az MPC a következő mikroutasításra mutat.Ha a JAM bitek közül 1 egy vagy több,akkor több munka szükséges.Ha a JAMN 1-es,az 1 bites N
flip-flop vagy-olódik(ORED) az MPC high-order(magasabbrendű) bitjében.Hasonlóan ha JAMZ
1-es,akkor az 1 bites Z flip-flop vagy-olódik azon a helyen.Ha mindkettő 1-es,mindkettő vagy-olódik azon a helyen.Az N és Z flip-flopok szükségességének oka a következő:az időtartamnak a növekvő széle után(amíg az óra magasan van) a B bus többé nem vezérelt,egyszóval az ALU kivezetések többé nem biztos,hogy helyesek.Elmentve az ALU státuszt N-be és Z-be,lehetővé tesszük a helyes adatokhoz jutást és az MPC számításához függetlenül attól mi történik az ALU körül.
A 4-6. ábrán a logikai egység az,ami a számítást végzi.
A Boolean függvény,amit kiszámít a következő:

F=(JAMZ és Z) vagy (JAMN és N) vagy NEXT_ADDRESS[8]

Vegyük észre,hogy minden esetben MPC-nek csak a legmagasabb helyiértéken vagyolva
2 értéke lehetséges:

1.NEXT_ADDRESS értéke.
2.NEXT_ADDRESS értéke a legmagasabb helyiértékű.

Nem létezik más lehetőség.Ha a NEXT_ADDRESS legmagasabb helyiértékű bit-je már 1 volt,
akkor a JAMN vagy JAMZ használata fölösleges.Vegyük észre,hogy amikor a JAM bitek
zéró értékűek ,a következő végrehajtandó mikroutasítás címe a 9 bites szám a NEXT_ADDRESS mezőben.Amikor JAMN vagy JAMZ 1,akkor két potenciális utód létezik:NEXT_ADDRESS és NEXT_ADDRESS vagy-olódás 0x100-zal(feltéve,hogy NEXT_ADDRESS

< vagy =,mint 0xFF).(0x mutatja,hogy a következő szám hexadecimális.)Ez illusztrálva a
4-7.ábrán látható.A folyamatban lévő mikroutasítások a 0x75 helyen van,
a NEXT_ADDRESS=0x92,és JAMZ 1-re van állítva.Tehát a
következő mikroutasítás címe az előző ALU műveleten tárolt Z bittől függ.
Ha a Z bit 0,akkor a következő mikroutasítás a 0x92 -ből jön.Ha 1,akkor 0x192-
ből jön.A JAM mezőn harmadik bit a JMPC.Ha egyes a 8 MBR bit 0RED a folyamatban
a NEXT_ADDRESS mezejének 8 low-order(alacsonyrendű) bitjével,
levő mikroutasítás.Az eredményt elküldik az MPC-nek.A 4-6.ábrán levő
'O'-val címzett doboz feladata,hogy készít egy OR-t(vagy-ot) MBR-rel és a NEXT_
ADDRESS-szel ha JMPC 1,de csak átadja NEXT_ADDRESS-t,az MPC-nek ha JMPC 0.
Ha JMPC 1,akkor a NEXT_ADDRESS low-order 8 bit-je egyértelműen zéró.A high-order
bit lehet 0 vagy 1,tehát a NEXT_ADDRESS értéke a JMPC-vel használva magától
értetődően 0x000 vagy 0x100.A 0x000 és a 0x100 használatát később tárgyaljuk.
217.o.
A lehetőség,hogy OR-t használhatunk az MBR és a NEXT_ADDRESS között és tárolhatjuk
az eredményt az MPC-ben lehetővé teszi egy sokirányú elágazásnak a megfelelő kivitelezését
(ugrásnak).Vegyük észre,hogy mind a 256 cím az MBR-ben levő
bitek által is meghatározható.Tipikus esetben az MBR tartalmaz egy opcode-ot
ezért a JMPC használata egyértelmű egyedi kiválasztását fogja eredményezni a következő
mikroutasításnak,minden lehetséges opcode-ba.A módszer
hasznos ha gyorsan,közvetlenül akarunk elágazni abba a funkcióba azt,amit
az éppen betöltött opcode definiál.A gép időszámlálását kritikus megérteni abból a
szempontból,hogy mi fog következni,ezért talán érdemes megismételni még egyszer.
Ezt alciklusos időintervallumokban csináljuk meg,ezt könnyű bemutatni,de az
érdekes események csupán a csökkenő szél ,amely a ciklust indítja,és az emelkedő
szél,mely betölti a regisztereket és az N és Z flip-flop-okat.Kérlek nézz
mégegyszer az 4-3.ábrára.Az 1.alciklus alatt az idő leeső élének
kezdeményezésére MIR feltöltődik a címről,amely az MPC-ben tárolódik.A 2.alciklus
alatt a MIR jelek kiáramlanak,a B bus pedig feltöltődik a választott regiszter-
ről.A 3.alciklus alatt az ALU és aléptető dolgozik és előállít egy stabil
eredményt.A 4.alciklus alatt a C bus,a memória bus-ok és az ALU értékek
stabilizálódnak.Az időtartam növekvő szélénél a regiszterek feltöltődnek a
C bus-ról,feltöltődnek N és Z flip-flop-ok,és az MBR és az MDR megkapja a
végeredményeit a memória műveletből,ami az előző adat út ciklus végén
indult(ha van ilyen).Amint az MBR elérhető,MPC feltöltődik felkészülve a
következő mikroutasításra.Így MPC megkapja az értékét valamikor az intervallum
közepén,amikor az időtartam magasan van,de csak miután MBR és MDR készen állnak.
Ez lehet szint által előidézett(inkább,mint szél által előidézett),vagy szél
által előidézett rögzített késleltetés az időtartam növekvő széle után.Az
egyetlen probléma,hogy MPC nincs feltöltve,míg a regiszterek nem állnak készen(MBR,N és

Z függvénye).Amint az idő lezuhan,MPC tudja címezni a mikroprogram tárt és egy új ciklus kezdődhet.

Vegyük észre,hogy mindegyik ciklus önmagát tartalmazza. Meghatározza,hogy mi megy a B bus-ra,mit csinálnak az ALU és a léptetők, hova kell tárolni a C buszt,és végül,hogy mi legyen a következő MPC értéke. Érdemes a 4-6. ábráról egy utolsó észrevételt tenni.Eddig úgy vettük MPC-t, mint egy igazi regisztert,9 bit tároló kapacitással,ami feltöltődik mialatt az időtartam...

***[218-221]

218
…magas. A valóságban egyáltalán nincs szükség regiszterekre. Az összes bemenete közvetlenül feltölthető a vezérlő tárolón keresztül. Amíg a vezérlő tárolóban vannak, az órajel leszállóágában van, és a MIR olvasás alatt áll, ez elegendő. Nincs szükség rá, hogy az MPC-ben tároljuk őket. Ezért MPC megvalósítható úgy mint egy **virtuális regiszter**, amelyik csak gyűjti a jeleket akár egy javító panel, nem pedig egy igazi regiszter. Az MPC virtuális regiszterré tétele egyszerűsíti az időzítést: az események már csak az órajel leszálló ás felszálló ágában történnek és máskor nem. Ha azonban az könnyebb, lehet az MPC-re mint valódi regiszterre is gondolni.

4.2 **EGY PÉLDA AZ ISA-RA: IJVM**

Folytassuk a példánkat a gép ISA szintjének bemutatásával, amelyet a mikroarchitektúrán futó mikroprogram értelmez, ahogy ezt a 4-6 ábra mutatja(IJVM). Néha úgy fogunk hivatkozni az ISA-ra mint **makroarchitektúrára**, hogy szemléltessük az eltérését a mikroarchitektúrától. Mielőtt meg tudnánk magyarázni, hogy mi is az az IJVM egy kissé el kell térnünk a témától.

4.2.1 **Vermek**

Virtuálisan minden programnyelvben vannak valamiféle eljárások(módszerek) amelyek saját változókat használnak. Ezeket el lehet érni az eljáráson belül, de a futása után már nem. Felmerül a kérdés: Hol legyenek ezek a változók a memóriában eltárolva?

A legegyszerűbb megoldás – hogy minden változónak külön abszolút memóriacímet adjunk – nem működik. A probléma abból adódik, ha az eljárás önmagát hívja. Ezeket a rekurzív eljárásokat majd az ötödik fejezetben tárgyaljuk bővebben. Egy pillanatra képzeljük el, hogy egy eljárás kétszer van meghívva. Ekkor lehetetlen lenne abszolút címen tárolni a lokális változóikat, hiszen a másodikként meghívott felülírná az előzőéit.

Ehelyett más módszert alkalmazunk. A memória egy területe – amit **verem**nek hívunk – van fenntartva ezen változók számára, de az egyes változóknak nincs abszolút memóriacímük benne. Ehelyett egy LV-nek nevezett regisztert alkalmazunk, amelynek az a feladata, hogy az éppen aktív eljárás helyi változóinak kezdetét tárolja le. A 4-8(a) ábrán egy A nevű eljárás meghívása történik, melynek lokális változói a1, a2 és a3, és ezek tárolására az LV-vel jelölt helyen kerül sor. Egy másik regiszter – az SP – A helyi változóinak legmagasabb szavára(word) mutat. Ha például LV 100 és a szavak 4 bájtosak, akkor SP 108 lesz. A változókra úgy hivatkozhatunk, hogy megadjuk relatív címüket(offset) LV-től. Az adatszerkezetet LV és SP között – ideértve a végpontokat is – A **helyi változó frame**jének nevezzük.

Most nézzük meg mi történik, ha A meghív egy B nevű eljárást. Hol lesznek B lokális változói(b1, b2, b3, b4) tárolva? A válasz: a veremben, A változói felett, ahogy a (b) ábra mutatja. Vegyük észre, hogy LV már B

219
első változójára mutat, A helyett. B helyi változói is megadhatók az LV-től számított eltolásukkal. Hasonlóan, ha B hívja C-t akkor LV és SP újra új értéket kapnak, ahogyan a (c) ábra mutatja.

Amikor C visszatér, visszaadja a vezérlést B-nek a verem visszaáll (b) állapotba, tehát LV újra B lokális változóira mutat. Ugyanígy, ha B futása véget ér visszajutunk az (a) ábrához. Tehát minden esetben LV az éppen aktív eljárás elejére, SP pedig a tetejére mutat.

Most tegyük fel, hogy A hívja D-t, és D-nek 5 lokális változója van. A (d) ábrán feltüntetett szituációhoz jutunk. Ekkor jól látható, hogy D változói azon a memóriaterületen kerültek eltárolásra ahol ezelőtt B és C változói voltak. Ezzel a memóriaszervezési módszerrel tehát csak az éppen aktív eljárásoknak van lefoglalva memóriaterület. Amikor az eljárás futása véget ér, a lokális változói által foglalt memória felszabadul.

A vermeknek van más felhasználási módjuk is a lokális változók tárolásán kívül. Felhasználhatók operandusok tárolására egy aritmetikai művelet elvégzése közben. Amikor így használjuk, a veremre **operandus verem**ként hivatkozunk. Vegyük példának azt az esetet, amikor A – mielőtt meghívná B-t – elvégzi a következő műveletet: a1=a2+a3. A feladatot megoldhatjuk úgy is, hogy a2-t letesszük(push) a verembe, ahogy azt a 4-9(a) ábra mutatja. Megjegyezzük, hogy az összes programrészt normál betűtípussal szedjük, ahogyan eddig. Ugyanezt használjuk az assembly nyelvű op-kódoknál, és gépi regisztereknél. Azonban a futtatási szövegben a program változói és eljárásai *dőlt* betűvel lesznek megadva. Az eltérés azért van, mert a változókat és eljárásokat a felhasználó választja meg, míg az opkódok és regiszterek nevei beépítettek.

Ekkor SP-t növeljük az egy szóban lévő bájtok számával – mondjuk néggyel – és az első operandust eltároljuk az így kapott SP címen. A következő lépésben a3-at tesszük le a verembe a (b) ábra szerint.

A művelet elvégezhető egy utasítás végrehajtásával, amely kiveszi(pop) két szót a veremből, összeadja őket, és az eredményt visszateszi oda, ahogy azt a (c) ábra mutatja. Végül a legfelső szó kivehető, és értékével feltölthető az a1 változó ((d) ábra).

220

A lokális változók és az operandusok tárolása egyszerre is megtörténhet. Például, ha egy olyan kifejezést kell kiszámolni mint x*x + f(x), akkor az x*x az operandus veremben kell hogy legyen mikor az f meghívódik. A függvény értéke pedig ott marad a veremben, hogy egy következő utasítás összeadhassa őket.

Megemlítésre érdemes, hogy minden gép használ vermeket lokális változók tárolására, míg csak kevés alkalmaz operandus vermeket. A JVM és az IJVM viszont az előbb bemutatott módon működik, ezért is mutattuk be ezt részletesen. (Az 5. fejezetben foglalkozunk velük részletesebben).

4.2.2 Az IJVM memória modell

Most már kész vagyunk, hogy megvizsgáljuk az IJVM felépítését. Olyan memóriából áll, ami kétféleképpen is vizsgálható: mint egy 4 294 967 296 bájtból (4GB) álló tömböt, vagy mint egy 1074 741 824 szóból állót, melynek minden egyes

eleme 4 bájtot tartalmaz. A legtöbb ISA-tól eltérően a JVM nem készít címeket közvetlenül a látható ISA szinten, annál inkább implicit(közvetett) címeket, amelyek a bázist biztosítják a mutatók számára. Az IJVM utasítások csak úgy érhetik el a memóriát, ha indexeli ezeket a mutatókat. A memóriának a következő részei vannak definiálva:

1. *A konstans tároló*. Ezt a területet nem írhatja IJVM program. Konstansokat, sztringeket és mutatókat tartalmaz (amelyek a memória elérhető részeire mutatnak). A programmal együtt töltődik a memóriába, és később nem változik. Itt van egy implicit regiszter, a CPP, amely a konstans tároló első szavának kezdőcímét tartalmazza.
2. *A lokális változó tároló*. Minden eljáráshíváskor egy memóriaterület van kijelölve a változók tárolására. Ennek a tárolónak az elején vannak a paraméterek(argumentumok) eltárolva. Ez a tároló nem tartalmazza az operandus vermet, ugyanis ez közvetlenül fölötte kezdődik. Van egy implicit regiszter, ami a helyi változók kezdőcímeit tárolja. Ezt a regisztert hívjuk LV-nek. Az eljárás hívásakor az előző paraméterek a lokális változó tár kezdeténél tárolódnak.

221

3. *Az operandus verem*. Nem léphet túl egy, a Java fordító által megadott méretet. Közvetlenül a lokális változó tároló fölött van elhelyezve a memóriában, ahogy azt a 4-10 ábra mutatja. A mi értelmezésünkben elegendő úgy gondolni az operandus veremre, mint a lokális változó tár egy részére. Akárhogy legyen is, létezik egy implicit regiszter, ami a verem legfelső szavának memóriacímét tartalmazza. Vegyük észre, hogy a CPP-től, és LV-től eltérően az SP megváltozik az eljárás futása alatt, ahogy az operandusokat leteszi, illetve felveszi a veremből.
4. *A metódus terület*. Végül van egy terület a memóriában, amely a programot tartalmazza. Erre úgy hivatkozunk a UNIX folyamatokban, mint 'text' területre. Van egy implicit regiszter, ami a következő utasítás címét tartalmazza. Ennek a mutatónak(pointer) a neve Program Counter, vagy PC. A memória többi részétől eltérően a metódus terület bájt tömbként van megadva.

A CPP, LV, és SP regiszterek szavakra mutatnak, nem bájtokra, és az eltolásuk(offset) a szavak számától függ. Az egész típusnak, amit választottunk az összes hivatkozása a konstans tárolóban, a lokális változó tárban, és a veremben szavakra történik, és az összes eltolás, ami ezekre a tárolókra vonatkozik szó. Például LV, LV+1, és LV+2 a lokális változó tár első három szavára mutatnak. Ugyanakkor LV, LV+4, LV+8 négy szavas(16 byte) intervallumokra mutatnak.

A PC viszont byte címeket tartalmaz, és növelésével vagy csökkentésével a cím bájtonként változik, nem pedig szavanként.

Fordította: Földesi Attila h938352

***[222-225]

Emlékezzünk, hogy ez csak 1 byte széles. A PC növelése eggyel és olvasás bevezetése a következő bájt előhozásához vezet. Az SP növelése eggyel és olvasás bevezetése a következő szó előrehozását eredményezi.

4.2.3 Az IJVM utasításkészlete

Az IJVM utasításkészlete a 4-11. ábrán látható. Minden utasítás egy opkódból és néha egy operandusból áll, csakúgy, mint egy memória offset vagy egy konstans. Az első oszlop adja az utasítás hexadecimális kódját, a második az assembly nyelvű mnemonikját, a harmadik pedig egy rövid működési leírást.

Jelentés

Betesz egy bájtot a verembe.
A verem legfelső szavát újra beírja a verembe.
Feltétel nélküli elágazás.
Fog két szót a veremből; Visszateszi az összegüket.
Fog két szót a veremből; Betesz egy logikai ést.
Fog egy szót a veremből és elágazik, ha 0.
Fog egy szót a veremből és elágazik, ha kisebb, mint 0.
Fog két szót a veremből; elágazik, ha egyenlőek.
Hozzáad egy konstanst egy lokális változóhoz.
Betesz egy lokális változót a verembe.
Meghív egy eljárást.
Fog két szót a veremből; betesz egy logikai vagyot.
Visszatér az eljárásból egy integer értékkel.
Vesz egy szót a veremből és eltárolja a lokákis vátozóban.
Vesz két szót a veremből;visszateszi a különbségüket.
Betesz egy konstanst a konstans területről a verembe.
Nem csinál semmit.
Törli a verem legfelső szavát.
Felcseréli a verem két legfelső szavát.
Előklépző utasítás; a következő utasítás egy 16 bites mutató.

4-11. ábra. Az IVJM utasításkészlete. Az operandusban szereplő byte, const és varnum egy bájtos., a disp, index és offset 2 bájtos.

Az utasítások képesek betenni a verembe egy szót különböző helyekről. Ilyen hely lehet a konstans terület (LDC_W), lokális váltózók területe (ILOAD), és maga az utasítás (BIPUSH). Egy változó ki is lehet venni a veremből és el lehet tárolni a lokális változók területén (ISTORE). Két számtani műveletet (IADD és ISUB), és ugyanúgy két logikai műveletet (IAND és IOR) lehet végrehajtani, ha a verem két legfelső szavát használjuk operandusként. Minden számtani és logikai műveletben, két szó vevődik ki a veremből, és az eredmény oda tevődik vissza. Négy elágazás utasítás van, egy feltétel nélküli (GOTO) és három feltételes (IFEQ, IFLT és IF_ICMPEQ). Minden elágazás utasítás, ha elindul, beállítja a PC értékét a saját (16 bittel jelölt) offsetjére, amit egy opkód követ az utasításban. Ez az offset hozzáadódik

az opkód címéhez. Vannak még IJVM utasítások, amelyek felcserélik a verem felső két szavát (SWAP), duplázzák a legfelső szót (DUP), és eltávolítják azt (POP).

Néhány utasításnak van többféle formája is, megengedve egy formát az általában használt verziónak. Az IJVM-be kétféle különböző JVM által használt mechanizmust tettünk be, ennek megvalósítására. Egy esetben kihagytuk a rövid formát egy sokkal általánosabb kedvéért. Egy másik esetben megmutattuk, hogy a WIDE előképző utasítás hogy tudja módosítani a következő utasítást.

Végül van egy utasítás (INVOKEVIRTUAL) egy másik eljárás segítségül hívására, és egy másik utasítás (IRETURN) az eljárásból való kilépésre és az irányítás visszaadására az eljárásnak ami meghívta ezt. A mechanizmus összetettsége miatt, kissé leegyszerűsítettük a definiciót, lehetségessé téve egy világos mechanizmus megteremtését a segítségül híváshoz és a visszatéréshez. A korlátozás az, hogy, nem úgy mint a Java-ban, mi csak egy eljárásnak engedjük meg, hogy segítségül hívjon egy eljárást a saját céljában létezve. Ez a korlátozás már komolyan megnyomorítja az objectumorientáltságot, de megengedi, hogy találjunk egy sokkal egyszerűbb módszert, elkerülve az eljárás dinamikus lokalizálásának követelményét. (Ha nem vagy otthonos az objektumorientált programozásban, nyugodtan feigyelmen kivül hagyhatod ezt a megjegyzést. Amit csináltunk az visszavezeti a Java-t egy nem objektumorientátl nyelvhez, mint a C vagy a Pascal). minden számítógépen, kivéve a JVM-et, a meghívandó utasítás címe közvetlenül a CALL utasításban határozódik meg, ezért a megközelítésünk valójában a normális eset nem a kivételes.

Az eljárást segítségül hívó mechanizmust a következők követik. Először a hívó betesz a verembe egy hivatkozási számot (mutatót) az objectumnak amit meg fogunk hívni. (Ez a hivatkozási szám nem szükséges az IJVM-ben, mivel nem lehet más objectumot megjelölni, de meg lett tartva a JVM-mel való következetesség miatt). A 4-12(a). ábrán ez a hivatkozási szám az OBJRF által van feltüntetve. Aztán a hívó rutin beteszi az eljárás parametereket a verembe, ebben a példában Paraméter 1, Paraméter 2 és Paraméte 3. Végül az INVOKEVIRTUAL elindul.

Az INVOKEVIRTUAL utasítás tartalmaz egy eltolódást, amelyik feltünteti a konstansterületbeni poziciót, ami magában foglalja a segítségül hívott eljárás eljárásterületbeni kezdőcímét. Azonban, amíg az eljáráskód az ezáltal a mutató által mutatott területen székel, az eljárásterület első négy bájtja speciális adatot tartalmaz. Az első 2 bájt 16 bites egészként van kiszámolva feltüntetve az eljáráshoz tartozó paraméterek számát (a paraméterek már előzőleg be lettek téve a verembe). Ehhez a számításhoz az OBJREF 0 paraméterű paraméterként lett számítva. Ez a 16 bites egészaz SP értékével együtt, az OBJREF helyét adja. Figyeljük meg, hogy az LV inkább az OBJREF-re mutat, mint az első valódi paraméter. A választás, hogy az LV hova mutasson, valamelyest önkényes.

Az eljárásterület második 2 bájtja egy másik 16 bites egészként számítódik ki, megadva az eljárásban használt lokális változóterület méretét. Ez feltétlenül szükséges, mert egy új verem lesz létrehozva az eljáráshoz, közvetlenül a lokális változóterület felett kezdődve. Végül, az ötödik bájt az eljárásterületen az első opkódot tartalmazza, ami el lesy indítva.

4-12. ábra. (a) A memória az INVOKEVIRTUAL futtatása előtt. (b) Futtatás után.

A valódi sorrend ami előfordul az INVOKEVIRTUAL-nál, úgy jön sorrendben, ahogy a 4-12. ábra mutatja. A két jelöletlen mutató bájt ami az opkódot követi, arra van használva, hogy készítsünk egy mutatót a konstansterület táblázatba (az első bájt

a magasszintű bájt).Az utasítás kiszámol egy alapcímet az új lokális változó területből, kivonva a paraméterek számát a vermmutatóbol, és beállítva LV-t, hogy az OBJREF-re mutasson. Ezen a helyen, felülírva OBJREF-et, a megvalósítás törli a címét annak a helynek, ahol a régi PC tárolva volt. Ez a cím úgy lesz kiszámolva, hogy a lokális vátozóterület mérete (parametere+lokális változók) hozzáadódik a címhez ami az LV-ben van. közvetlenül a cím felett, ahol a a régi PC lett eltárolva, az a cím van, ahol a régi LV lett eltárolva. Közvetlenül e cím felett van az újonnan meghívott eljáráshoz tartozó verem kezdete. Az SP úgy van beállítva, hogy a régi LV-re mutasson, ami közvetlenül az első hely alatt lévő cím a veremben. Emlékezzünk arra, hogy az SP mindig a verem legfelső szavára mutat. Ha a verem üres, akkor a verem vége alatti első helyre mutat, mert a vermünk felfelé növekszik, a magasabb című területek felé. Az ábránkon a verem mindig felfelé növekszik a magasabb című területek felé a lap tetején.

Az utolsó műveletre azért van szükség, hogy végrehajtsa az INVOKEVIRTUAL-t, beállítja a PC-t, hogy az eljárás kód terület ötödik bájtjára mutasson.

4-14. ábra. (a) A memória az IRETURN futtatása előtt. (b) Futtatása után.

Az IRETURN utasítás az INVOKEVIRTUAL fordítottja, ahogy az a 4-13. ábrán látszik. Felszabadítja a visszatérési eljárás által használt területeket. A vermet is visszaállítja az előző helyzetébe, kivéve, hogy (1) az (már felülírt) OBJREF szó és minden paraméter ki lett szedve a veremből, és (2) a visszatérési érték a verem tetején fog elhelyezkedni, azon a helyen, ahol előzőleg az OBJREF foglalt el. Hogy visszaállítsa az előző helyzetet, az IRETURN képes kell legyen arra, hogy visszaállítsa a PC és LV mutatókat a régi értékeikre. Ezt úgy csinálja, hogy megcímezi a link mutatót (amelyik az a szó, amit az aktuális LV mutató azonosít). Ezen a helyen, emlékezzünk vissza, ahol eredetileg az OBJREF volt eltárolva, az INVOKEVIRTUAL utasítás tárolta el a régi PC-t tartalmazó címet. Ez a szó, és az e felett lévő szó visszakeresi a PC és LV visszaállítását, illetve a régi értékeiket. A visszatérő érték, ami a lezáró eljárás veremjének a tetején helyezkedik el, átmásolódik arra a helyre, ahol eredetileg az OBJREF volt tárolva, és az SP visszaállítódik, hogy erre a helyre mutasson. Az irányítás ezért fog arra az utasításra visszatérni, amelyik közvetlenül követi az INVOKEVIRTUAL utasítást.

***[226-229] javított!

Eleddig a számítógépünk nem tartalmaz I/O utasításokat. Mi sem deklarálunk. Nincs szüksége többre, mint amennyire a Java Virtual Machine-nek, és a JVM hivatalos specifikációja sehol sem említi az I/O -t. Az elmélet az, hogy az a számítógép, mely nem tartalmaz sem bemenetet, sem kimenetet, "biztonságos". (Az írást és olvasást a JVM speciális módszerek hívásával végzi, amelyek helyettesítik az I/O-t.)

4.2.4 A Java kód fordítása IJVM-é

Nézzük, hogyan kapcsolódik a Java és az IJVM egymáshoz. A 4-14-es ábrán egy egyszerű Java kód töredékét mutatjuk be. Ha ezt egy Java fordítóba töltenénk, akkor a fordító valószínűleg a 4-14(b) ábrán bemutatott IJVM assembly nyelvet készítené el. Az assembly kód bal oldalán álló 1-től 15-ig terjedő sorszámozás nem része a fordító kimenetének. A megjegyzések sem amelyek //-el kezdődnek. Azért vannak ott, hogy segítsenek megérteni egy részfolyamatot. A Java assembler ezután átfordítaná az assembly programot a 4-14(c) ábrán bemutatott bináris programba. (Egyébként a Java fordító elkészíti a saját assembler kódját, és közvetlenül bináris programba fordít.) Ebben a példában feltettük, hogy i 1 sorszámú lokális változó, j 2 sorszámú lokális változó, és k 3 sorszámú lokális változó.

A lefordított kód magtáól értetődő. Először j-t és k-t a verembe teszi, összeadja, és az eredményt i-ben tárolja. Ezután i-t és a 3 értékű konstanst helyezi a verembe, majd összehasonlítja őket. Ha egyenlők, akkor azon az ágon a vezérlést az L1 címkére adja át, ahol k értékét 0-ra állítja. Ha különbözőek,

akkor az összehasonlítás sikertelen és a kódban közvetlenül következő IF_ICMPEQ hajtódik végre. Ha ez befejeződött, akkor ez az ág az L2 címkénél folytatódik, ahol a then és else részek egyesülnek.

Az 4-14(b) ábra IJVM programjának operandus vermét a 4-15 ábrán mutatjuk be. Mielőtt a kód futása elkezdődik, a verem üres, amit a 0 feletti vízszintes vonal jelöl. Az első ILOAD után a j a veremben van, ahogyan azt az 1-es (tehát: első utasítás végrehajtódott) feletti j is jelzi. A második ILOAD után két szó van a veremben, ahogyan az a 2-es felett látszik. Az IADD utasítás után csak egy szó marad a veremben, mely a j+k összeget tartalmazza. Amikor a verem legfelső szavát kiemeljük, és i-ben tároljuk, a verem kiürül, ahogy az a 4-es felett látszik.

Az 5. utasítás (ILOAD) kezdi az if utasítást az i verembe helyezésével (5-ös). Ezután jön a 3 értékű konstans (6-os). Az összehasonlítás után a verem ismét üres (7-es). A 8. utasítás a Java else (maradék) része. Az else rész a 12. utasításig folytatódik, ekkor átlép a then részen és az L2 címkére ugrik.

4.3 Egy példa-implementáció

Miután részletesen bemutattuk a mikroarchitektúrát és a makroarchitektúrát, a következő téma az implementáció. Más szavakkal, hogyan néz ki egy program, mely az elsőn (mikroarchitektúra) fut, a másikat interpretálja, és hogyan működik? Mielőtt megválaszolhatnánk ezeket a kérdéseket, gondosan át kell gondolnunk a jelölést, amelyet az implementáció leírására használunk.

4.3.1 Mikroutasítások és jelölés

Először is, leírhatnánk a vezérlő tárat binárisan is, 36 biten szavanként. De – mint a hagyományos programozási nyelvekben – sokat nyerünk, ha bevezetünk egy jelet, amely tartalmazza az általunk vizsgált probléma lényegét, miközben elrejti azokat a részeket, melyeket figyelmen kívül hagyhatunk, vagy később könnyebben kezelhetünk automatikusan. Fontos, hogy itt tisztában legyünk azzal, hogy a nyelv, melyet választottunk, inkább arra szolgál, hogy illusztráljuk az elveket, semmint megkönnyítsünk egy hatékony tervezést. Ha ez utóbbi lenne a célunk, akkor egy gyökeresen különböző jelölést használnánk, hogy a lehető legrugalmasabb rendszer álljon a tervező rendelkezésére. Az egyik szempont, mely szerint ez a téma fontos, az a címek kiválasztása. Mivel a memória nincs logikai sorrendben, nem természetes a "következő utasítás" beillesztése, miközben az utasítások sorrendjét határozzuk meg. Az ilyen vezérlés-szervezésnek legnagyobb előnye abból származik, hogy a tervező (vagy az összeállító) hatékonyan megválaszthatja a címeket. Mi ezért egy egyszerű szimbolikus nyelv bevezetésével kezdünk, amely egészében leír minden utasítást, anélkül, hogy részletesen kifejtené, hogyan határozza meg az összes címet.

Jelölésünk meghatároz minden tevékenységet, amely egy órajel alatt az adattárolóban történik. Elméletileg használhatnánk egy magas-szintű nyelvet az utasítások leírására. Mindemellett az órajelről-órajelre vezérlés nagyon fontos, mert lehetőséget biztosít egyidejűleg több utasítás használatára, és fontos, hogy képesek legyünk megvizsgálni minden órajelet, hogy megértsük és ellenőrizzük az utasításokat. Ha a cél egy gyors, hatékony implementáció (ha a többi dolog megegyezik, gyors és hatékony mindig jobb, mint lassú és nehézkes), akkor minden órajel számít. Egy valódi implementáció esetén sok körmönfont trükk van elrejtve a programban, jelentéktelen szekvenciákat vagy utasításokat használva, hogy akár egyetlen órajelet megtakarítson. Nagyon megéri az órajelekkel takarékoskodni: ha adott egy négy órajeles utasítás, amelyet kettővel csökkenhetünk, akkor így már kétszer gyorsabban fut. Ezt a többletteljesítményt pedig az utasítás minden hívásakor megkapjuk.

Az egyik lehetséges megközelítés egyszerűen kilistázni

azokat a jeleket, melyeket egy-egy órajelben aktiválni akarunk. Tegyük fel, hogy egy órajel alatt növelni akarjuk SP értékét. Ezenkívül el akarunk indítani egy read műveletet is, és azt akarjuk, hogy a következő utasítás a vezérlő rekesz 122-es helyén legyen található. Talán az írjuk:

ReadRegister=SP, ALU=INC, WSP, Read, NEXT_ADDRESS=122

ahol a WSP “az SP regiszter írás”-át jelenti. Ez a jelölés teljes, de nehezen értelmezhető. Ehelyett kombinálni fogjuk az utasításokat egy természetes és intuitív módon, hogy visszaadjuk, mi is történik valójában:

SP=SP+1; rd

Nevezzük magas szintű Micro Assembly Nyelvünket MAL-nek (francia szó, jelentése: beteg; valami, amivé válsz ha túl sok kódot kell ebben írnod). A MAL-t úgy terveztük, hogy visszaadja a mikroarchitektúra karakterisztikáját. Az egyes órajelek alatt bármely regiszter írható, de általában csak egyet írunk. Csak egy regiszter juttatható az ALU B oldalára. Az A oldalon a +1, 0, -1 és a H regiszter választható. Így használhatunk egy egyszerű értékadó utasítást, mint Javaban, hogy feltüntessük a végrehajtandó műveletet. Például az SP-ből MDR-be való másolásra mondhatjuk:

MDR=SP

Az ALU funkcióinak kihasználásához – a B buszon való áthaladáson kívül –, írhatjuk például:

MDR=H+SP

Ez a H regiszter tartalmát hozzáadja SP-hez és az eredményt MDR-be írja. A + operátor kommutatív (ami azt jelenti, hogy az operandusok sorrend nem számít), így a fenti állítás így is írható:

MDR=SP+H

és ugyanazt a 36 bites mikroutasítást állítja elő, azt is beleértve, hogy H szigorúan a bal ALU operandus.

Vigyáznunk kell, hogy csak szabályos utasításokat használjunk. A legfontosabb ilyen utasítások a 4-16-os ábrán találhatóak, ahol SOURCE bármely lehet az MDR, PC, MBR, MBRU, SP, LV, CPP, TOS vagy OPC (MBRU jelöli az MBR előjel nélküli verzióját). Ezek a regiszterek mind lehetnek az ALU forrásai a B buszon. Hasonlóan DEST csak a MAR, MDR, PC, SP, LV, CPP, TOS, OPS vagy H regiszterek valamelyike lehet, közülük pedig mind lehet célja az ALU C buszra menő outputjának. Ez a forma megtévesztő, mert sok látszólag ésszerű utasítás szabálytalan. Például a

MDR=SP+MDR

tökéletesen ésszerűnek látszik, de nincs lehetőség rá, hogy a 4-16 ábra szerinti adat-útvonalon egy órajel alatt végigfusson. Ez a megszorítás azért van mert egy összegzésben (nem úgy, mint egy növelésben vagy csökkentésben) az egyik operandus a H kell, hogy legyen. Ezért a

H=H-MDR

bár hasznos lenne, szintén megvalósíthatatlan, mert a kivonandó (amit kivonunk) egyetlen lehetséges forrása a H regiszter. Az assembler dolga, hogy visszautasítsa az állításokat, amelyek jóknak tűnnek, ám valójában szabálytalanok.
Kiterjesztjük a jelölét, hogy megengedje a többszörös értékadást az egyenlőség-jelek többszöri használatával. Például SP-hez 1-et adni, majd ezt SP-n tárolni és MDR-be írni megoldható így:

SP=MDR=SP+1.

Hogy jelezzük, hogy a memória 4-bájtos adatszavakat ír és olvas, csak rd-t vagy wr-t teszünk a mikroutasításokba. Egy bájt átjuttatását az 1-bájtos porton fetch jelöli. A hozzárendelések és a memória-utasítások megjelenhetnek ugyanabban az órajelben is. Ezt úgy jelöljük, hogy egy sorba írjuk őket.

Hogy elkerüljük a félreértéseket, hadd szögezzük le ismét: a Mic-1 két úton éri el a memóriát. Ír és olvas 4-bájtos adatszavakat MAR/MDR-t használva, és a mikorutasításokban rw-vel vagy rd-vel jelölve, külön-külön. 1-bájtos utasításkódokat olvas az utasítás stream-ből PC/MBR-t használva, amit fetch-el jelöl a mikroutasításokban. Mindkét memóriaművelet folyhat egyszerre.

Ezzel együtt ugyanaz a regiszter nem kaphat értéket a memóriából és az adat-útvonalról is egy órajelen belül. Figyeljük meg a következő kódot:

MAR=SP; rd
MDR=H

Az első mikroutasítás hozzárendel MDR-hez egy értéket a memóriából a második mikroutasítás végén.

***[230-233]
Nem érkezett meg! Csaba Gábor: h938343

***[234-237] javított!

234. ábra
4. fejezet A mikroprogramok szintje
4-17. ábra A Mic-1 mikroprogramja (1. rész)
cimke műveletek magyarázat
Main1 PC=PC+1; fetch; goto(MBR) MBR tárolja a műveleti kódot; következő byte
beolvasása; elküldés(dispatch)
Nop1 goto Main1 Nem tesz semmit.
Iadd1 MAR=SP=SP-1; rd Beolvassa a verem legfelső alatti szavát.
Iadd2 H=TOS H=a verem teteje
Iadd3 MDR=TOS=MDR+H; wr; goto Main1 Összeadja a két felső szót; beírja a verem tetejébe
Isub1 MAR=SP=SP-1; rd Beolvassa a verem legfelső alatti szavát
Isub2 H=TOS H=TOS (top of verem-a verem legfelső szava)
Isub3 MDR=TOS=MDR-H; wr; goto Main1 Elvégzi a kivonást; beírja a verem tetejébe
Iand1 MAR=SP=SP-1; rd Beolvassa a verem legfelső alatti szavát
Iand2 H=TOS H=TOS
Iand3 MDR=TOS=MDR AND H; wr; goto Main1 “És” művelet végrehajtása; beírja az új verem tetejébe
Ior1 MAR=SP=SP-1; rd Beolvassa a verem legfelső alatti szavát
Ior2 H=TOS H=verem teteje
Ior3 MDR=TOS=MDR OR H; wr; goto Main1 “Vagy” művelet végrehajtása; beírja a verem új
tetejébe
Dup1 MAR=SP=SP+1 Növeli SP értékét és bemásolja MAR-ba
Dup2 MDR=TOS; wr; goto Main1 A verem új szavának beírása
Pop1 MAR=SP=SP-1;rd Beolvassa a verem legfelső alatti szavát
Pop2 Vár, amíg a memóriából betöltődik az új TOS
Pop3 TOS=MDR; goto Main1 Az új szó beolvasása a TOS-ba
Swap1 MAR=SP-1; rd MAR értékét SP-1-re állítja; a verem második
szavát beolvassa
Swap2 MAR=SP A MAR-t a legfelső szóra állítja
Swap3 H=MDR; wr TOS-t elmenti H-ba; a második szót beírja
TOS-ba
Swap4 MDR=TOS A régi TOS-t bemásolja MDR-be

Swap5	MAR=SP-1; wr	MAR értékét SP-1-re állítja; a verem második szavaként írja be
Swap6	TOS=H; goto Main1	TOS felülírása (update)
Bipush1	SP=MAR=SP+1	MBR=a verembe illesztendő byte
Bipush2	PC=PC+1; fetch	PC értékét növeli; következő műveleti kód beolvasása
Bipush3	MDR=TOS=MBR; wr; goto Main1	A konstans előjeles kiterjesztése; verembe beillesztése
Iload1	H=LV	MBR tartalmazza az indexet; LV-t bemásolja H-ba
Iload2	MAR=MBRU+H; rd	MAR=a beillesztendő helyi változó címe
Iload3	MAR=SP=P+1	SP az új verem tetejére mutat; írás előkészítése
Iload4	PC=PC+1; fetch; wr	PC-t növeli; következő műv. kód beolvasása; verem-tető átírása
Iload5	TOS=MDR, goto Main1	TOS felülírása
Istore1	H=LV	MBR tartalmazza az indexet; LV-t bemásolja H-ba
Istore2	MAR=MBRU+H	MAR=azon helyi változó címe, ahova menteni akarunk
Istore3	MDR=TOS; wr	TOS bemásolása MDR-be; szó írása
Istore4	SP=MAR=SP-1; rd	Beolvassa a verem legfelső alatti szavát
Istore5	PC=PC+1; fetch	PC-t növeli; következő műveleti kód beolvasása
Istore6	TOS=MDR; goto Main1	TOS felülírása
Wide1	PC=PC+1; fetch; goto (MBR OR 0x100)	Többszörös elágazás felső bitbeállításával
Wide-iload1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Wide-iload2	H=MBRU<<8	H=az első index byte 8 bittel balra tolva
Wide-iload3	H=MBRU OR H	H=a helyi változó 16 bites indexe
Wide-iload4	MAR=LV+H; rd; goto iload3	MAR=a beillesztendő helyi változó címe
Wide-istore1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Wide-istore2	H=MBRU<<8	H=az első index byte 8 bittel

balra tolva

Wide-istore3	H=MBRU OR H	H=a helyi változó 16 bites indexe
Wide-istore4	MAR=LV+H; goto istore3	MAR=a helyi változó címe, ahol tárolni akarunk
Ldc_w1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Ldc_w2	H=MBRU<<8	H=első index byte<<8
Ldc_w3	H=MBRU OR H	H=16 bites indexet a konstans-készletbe
Ldc_w4	MAR=H+CPP; rd; goto iload3	MAR=a konstans készlet címe

cimke	műveletek	magyarázat
iinc1	H=LV	MBR tárolja az indexet; LV-t bemásolja H-ba
iinc2	MAR=MBRU+H; rd	LV+index értéket bemásolha MAR-ba; változó beolvasása
iinc3	PC=PC+1; fetch	Konstans beolvasása
iinc4	H=MDR	Változó bemásolása H-ba
iinc5	PC=PC+1; fetch	Következő műveleti kód beolvasása
iinc6	MDR=MBR+H; wr; goto Main1	Összeget bemásolja MDR-be; változó felülírása
goto1	OPC=PC-1	A műveleti kód címének eltárolása
goto2	PC=PC+1; fetch	MBR=az eltolási cím első byte-ja; második byte beolvasása
goto3	H=MBR<<8	Emelés és a megjelölt első byte H-ba mentése
goto4	H=MBRU OR H	H=16 bites elágazás eltolási cím
goto5	PC=OPC+H; fetch	Eltolási cím hozzáadása OPC-hez
goto6	goto Main1	Várakozás a következő műveleti kód beolvasására
iflt1	MAR=SP=SP-1; rd	A verem legfelső alatti szavának beolvasása
iflt2	OPC=TOS	TOS ideiglenes tárolása OPC-ben
iflt3	TOS=MDR	Az új verem tetejének tárolása TOS-ba
iflt4	N=OPC; if(N) goto T; else goto F	N bitre történő elágazás (Branch on N bit)
ifeq1	MAR=SP=SP-1; rd	A verem legfelső alatti szavának beolvasása
ifeq2	OPC=TOS	TOS ideiglenes tárolása OPC-ben

ifeq3 TOS=MDR Az új veremadat tárolása TOS-
ba
ifeq4 Z=OPC; if(Z) goto T; else goto F Z bitre történő elágazás (Branch
on Z bit)
if_icmpeq1 MAR=SP=SP-1; rd A verem legfelső alatti szavának
beolvasása
if_icmpeq2 MAR=SP=SP-1 MAR beállítása az verem új
tetejének beolvasására
if_icmpeq3 H=MDR; rd A verem második szavának
bemásolása H-ba
if_icmpeq4 OPC=TOS TOS ideiglenes tárolása OPC-
ben
if_icmpeq5 TOS=MDR Az verem új tetejének tárolása
TOS-ba
if_icmpeq6 Z=OPC-H; if (Z) goto T; else goto F Ha a két felső szó megegyezik,
menjen T-be,

egyébként F-be
T OPC=PC-1; fetch; goto goto2 A goto1-hez hasonló, a célzott
címhez szükséges
F PC=PC+1 Az eltolási cím első byte-jának
kihagyása
F2 PC=PC+1; fetch PC most a következő műveleti
kódra mutat
F3 goto Main1 Várakozás a műveleti kód
beolvasására
Invokevirtual1PC=PC+1; fetch MBR=1-es indexbyte; PC
növelése; 2. byte beolvasása
Invokevirtual2H=MBRU<<8 Emelés, és első byte tárolása H-
ba
Invokevirtual3H=MBRU OR H H=a program-mutató eltolási
címe CPP-ből
Invokevirtual4MAR=CPP+H; rd A metódus-mutató beolvasása a
CPP körzetből
Invokevirtual5OPC=PC+1 Visszatérési PC ideiglenes
tárolása OPC-ben
Invokevirtual6PC=MDR; fetch PC az új programra mutat;
paraméter számláló

beolvasása
Invokevirtual7PC=PC+1; fetch A paraméter számláló második
byte-jának beolvasása
Invokevirtual8H=MBRU<<8 Emelés és első byte mentése H-
ba
Invokevirtual9H=MBRU OR H H=a paraméterek száma
Invokevirtual10 PC=PC+1 # helyváltozók első
byte-jának beolvasása (# locals)
Invokevirtual11 TOS=SP-H TOS=OBJREF-1
címe
Invokevirtual12 TOS=MAR=TOS+1
TOS=OBJREF címe (új LV)

Invokevirtual13	PC=PC+1; fetch	# hely második byte-jának beolvasása (# locals)
Invokevirtual14	H=MBRU<<8	Emelés/léptetés és az első byte mentése H-ba
Invokevirtual15	H=MBRU OR H	H=# helyi változók (locals)
Invokevirtual16	MDR=SP+H+1; wr	OBJREF felülírása a link-mutatóval
Invokevirtual17	MAR=SP=SP+1	SP, MAR beállítása arra a helyre, ahova a régi PC-t mentjük
Invokevirtual18	MDR=LV; wr	A régi PC mentése a helyi változók fölé
Invokevirtual19	MAR=SP=SP+1	SP arra a helyre mutat, ahol a régi LV tároljuk
Invokevirtual20	MDR=LV; wr	Régi LV mentése a mentett PC fölé
Invokevirtual21	PC=PC+1; fetch	Az új metódus első műveleti kódjának beolvasása
Invokevirtual22	LV=TOS; goto Main1	LV-t beállítja, hogy LV területére mutasson

cimke	műveletek	magyarázat
ireturn1	MAR=SP=LV; rd	SP, MAR nullázása az új link-mutató beolvasásához
ireturn2		Várakozás az olvasásra
ireturn3	LV=MAR=MDR; rd	LV beállítása a link-mutatóra; régi PC beolvasása
ireturn4	MAR=LV+1	MAR beállítása a régi LV beolvasásához
ireturn5	PC=MDR; rd; fetch	PC visszaállítása; következő műveleti kód beolvasása
ireturn6	MAR=SP	MAR beállítása a TOS írására
ireturn7	LV=MDR	LV visszaállítása
ireturn8	MDR=TOS; wr; goto Main1	A visszaadott érték mentése az eredeti veremtetőbe

4-17. ábra A Mic-1 mikroprogramja (3/3)

Ha az MBR-ben lévő byte csupa 0, a NOP utasításhoz tartozó műveleti kód a következő, úgy a következő mikroutasítás nop1-gyel jelölt lesz, és a 0. rekeszből lesz beolvasva . Minthogy ez az utasítás nem tesz semmit, egyszerűen visszaugrik a fő ág kezdetére, ahol a ciklus megismétlődik, de egy új MBR-be olvasott műveleti kóddal.

Újra kihangsúlyozzuk, hogy a mikroutasítások a 4-17. ábrán a memóriában nem egymás után következnek, és hogy a Main1 nem a nullás vezérlő tár címen van (mert a nop1-nek kell a nullás címen lennie). A mikroassembler végzi el minden egyes mikroutasítás megfelelő helyre történő elhelyezését és rövid sorozatokba történő rendezését a NEXT_ADDRESS mező felhasználásával. Minden sorozat az általa interpretált IJVM műveleti kód numerikus értékének megfelelő címen kezdődik (pl.

POP a 0x57-es címen indul), de a ciklus többi része bárhol lehet a vezérlés tárolóban, és nem feltétlenül egymást követő címeken.

Most vizsgáljuk meg a IJVM IADD utasítását. A ciklusból az iadd1-gyel jelölt mikroutasításokhoz kell jutnunk. Ez az utasítás az IADD-ra jellemző műveletettel kezdődik:

1. A TOS már létezik, de a verem legfelső alatti szavát be kell olvasni a memóriából.
2. A TOS-t hozzá kell adni a memóriából behozott legfelső alatti szóhoz.
3. Az eredményt, amelyet be kell írni a verembe, el kell tárolni a memóriába, minthogy a TOS regiszterbe is.

Az operandus memóriából való betöltéséhez szükséges a verem mutató csökkentése, és annak beírása a MAR-ba. Meg kell jegyeznünk, hogy ez a cím egyben az is, melyet a következő írásnál használunk. Továbbá mivel ez a hely lesz a verem teteje, SP-hez is hozzá kell rendelni ezt az értéket. Ezért egyetlen művelet meg tudja állapítani SP és MAR új értékét, csökkentheti SP-t, és azt mindkét regiszterbe beírhatja.

Ezek az első műveletben mennek végre, (iadd1) és az olvasási művelet elindulnak. Emellett MPC megkapja az értéket iadd1 NEXT_ADDRESS mezőjéről, amely iadd2 helye, bárhol is van. Ekkor iadd2 be lesz olvasva a vezérlő tárból. A második ciklusban, mialatt várunk az operandusnak a memóriából történő beolvasására, bemásoljuk a verem legfelső szavát a TOS-ból H-ba, ahol elérhető lesz az összeadáshoz, mikorra a beolvasás befejeződik.

A harmadik ciklus (iadd3) elején, MDR tartalmazza a memóriából beolvasott összeadandót. Ebben a ciklusban ez hozzáadódik a H tartalmához, és az eredmény visszaíródik MDR-be, valamint a TOS-ba. Egy írási művelet is elindul, amely a verem tetejét elmenti a memóriába. Ebben a ciklusban a GOTO-nak a hatása az, hogy hozzáfűzi a Main1 címét az MPC-hez, eljuttatva minket a következő utasítás végrehajtásának kezdőpontjához.

Ha a következő műveleti kód, amit most az MBR tárolódik, 0x64 (ISUB), a műveleteknek majdnem pontosan ugyanaz a sorrendje alakul ki ismét. A Main1 elindulása után a vezérlés áttevődik a 0x64-en lévő mikroutasításnak (isub1). Ezt a mikroutasítást az isub2, isub3 és végül ismét Main1 követi. Az egyetlen különbség az előző és a mostani sorrend között, hogy az isub3-ban a H tartalmát kivonjuk MDR-ből az összeadás helyett.

Az IAND értelmezése majdnem azonos az IADD, és ISUB-éval, kivéve, hogy a verem két felső szava bitenként “és”-elődik, összeadás vagy kivonás helyett. Hasonló dolog történik az IOR-ral.

Ha az IJVC műveleti kód értéke DUP, POP vagy SWAP, a vermet meg kell változtatni. A DUP utasítás egyszerűen megismétli a verem tetejét. Mivel e szó értéke már tárolva van a verem tetején, a művelet ugyanolyan egyszerű, mint SP növelése, úgy, hogy az új helyre mutasson, és elmenteni TOS-t erre a helyre. A POP utasítás is majdnem ilyen egyszerű, csak csökkenti SP-t, a verem legfelső szavának törléséhez. Azonban a verem tetejének megtartásához most szükséges a verem új tetejének beolvasása a memóriából, és annak TOS-ba való írása. Végül, a SWAP utasítás magába foglalja a memória két helyén lévő értékek cseréjét: a verem két legfelső szaváak cseréjét. Ez valamivel könnyebb, mivel a TOS már tartalmazza ezen értékek egyikét, ezért ezt nem kell a memóriából beolvasni. Az utasítást a későbbiekben részletesebben tárgyaljuk.

A BIPUSH instrukció ennél kicsit komplikáltabb, mivel az utasítás kódját egy egyszerű byte követi, ahogy az a 4-18. ábrán látható. A byte-ot előjeles egészként kell értelmezni. Ezt a byte-ot, amely már betöltődött a memóriába a Main1 során, előjelesen ki kell terjeszteni 32 bitesre, és be kell illeszteni a verem tetejébe. Ez a művelet ezért ki kell, hogy terjessze előjelesen az MBR-ben levő byte-ot 32 bitessé, és be kell másolja MDR-be. Végül SP értéke nő eggyel, és be lesz másolva MAR-ba, engedélyezve az operandus kiíratását a verem tetejébe. Ezalatt ennek az operandusnak a TOS-ba is be kell másolódnia. Tehát, a főprogramhoz való visszatérés előtt, a PC értékét növelni kell, hogy a következő műveleti kód elérhető legyen a Main1-ben.

4-18. ábra A BIPUSH utasítás formátuma

***[238-241] javított!

Dani Andrea
KPM I.

Következőnek tekintsük az ILOAD utasítást. Ennél az utasításnál is egy byte követi az utasítás azonosítóját (opcode), mint azt a 4-19 (a) ábra mutatja, de ez a byte egy (előjel nélküli) index, amely azt a szót azonosítja a lokális változók memóriaterületén, amelyet a verem tetejére kell tenni (push onto the stack). Mivel csak 1 byte áll rendelkezésre, csak 2^8=256 szót lehet megkülönböztetni (elérni), mégpedig az első 256 szót a lokális változók memóriaterületén. Az ILOAD utasítás megkövetel mind egy olvasást (read) (hogy megkapják a szót) és egy írást (write) (hogy a verem tetejére tudjuk tenni azt). Mindazonáltal, hogy az olvasandó címet meghatározzuk, az eltolást (offset), amely az MBR-ben található, hozzá kell adnunk az LV tartalmához. Mivel mind az MBR mind az LV csak a B buszon keresztül érhető el, ezért LV-t először H-ba másoljuk (iload 1-ben), majd hozzáadjuk MBR-et. Ezen összeadás eredménye ezután MAR-ba másolódik és megkezdődhet a read (iload 2-ben).

4-19. (a) 1 byte-os indexű ILOAD. (b) 2 byte-os indexű WIDE ILOAD

Megjegyzendő azonban, hogy az MBR-nek indexként való használata kissé különbözik attól, amit a BIPUSH-nál láthattunk, ahol az előjelesen volt értelmezve. Index esetén az offset mindig pozitív, tehát az 1 byte-os offset-et egy előjel nélküli egészként kell kezelni, ellentétben a BIPUSH-al, ahol az egy előjeles 8 bites egészként volt interpretálva. Az interface az MBR-től a B busz felé nagyon gondosan lett megtervezve, hogy mindkét művelet lehetséges legyen . A BIPUSH esetén (előjeles 8 bites egész) a megfelelő művelet az elöjel-kiterjesztés, azaz a “legbaloldalibb” bitje az 1 byte-os MBR-nek bemásolódik a B busz felső 24 bitjébe. Az ILOAD esetén (előjel nélküli 8 bites egész), a megfelelő művelet a zéró-kitöltés (zero-fill). Itt a B busz felső 24 bitje egyszerűen nullákkal van feltöltve. Ezek a műveletek különböző jelekkel vannak megkülönböztetve, amelyek jelzik, hogy melyiket kell végrehajtani (ld. a 4-6 ábrát). A mikrokódban ezt úgy jelöljük, hogy MBR-t írunk (előjeles esetben, mint pl a BIPUSH 3) vagy MBRU-t (előjeltelen esetben, mint az iload 2).
Míg memóriára várunk, amely biztosítja az operandust (iload 3-ban), SP értéke megnő eggyel, hogy helyesen tartalmazza az eredményt, az új veremtetőt (Top of Stack, TOS). Ez az érték is MAR-ba másolódik előkészületként ahhoz, hogy az operandust kiírhassuk az új veremtetőbe. A PC-t újra meg kell növelni, hogy behozhassuk a következő utasítás kódját (iload 4- ben). Végül MDR értéke TOS-be másolódik, hogy mutassa az új veremtető értékét (iload 5-ben).

ISTORE az inverz művelete ILOAD-nak, azaz egy szót eltávolítunk a verem tetejéről és eltároljuk azon területen, amit az LV és az utasítás indexének az összege határoz meg. Ugyanazt a formátumot használja, mint az ILOAD, amelyet a 4-19 (a) ábrán láthatunk, azzal a különbséggel, hogy az opcode 0x36-os a 0x15 helyett. Ez a művelet kissé különbözik attól, mint amit várnánk, mert a verem tetején levő szó már ismert (TOS-ben található) így egyből el lehet tárolni. Mindazonáltal az új veremtető-szót be

kell tölteni. Így mind egy read mind egy write szükséges, de ezek akármilyen sorrendben végrehajthatók (vagy akár még párhuzamosan is, ha ez lehetséges).
Mind az ILOAD, mind az ISTORE korlátozva vannak azáltal, hogy csak az első 256 lokális változót tudják elérni.. Míg a legtöbb program esetén mindössze ennyi memóriaterületre van szükség a lokális változók részére, természetesen szükséges lehet elérni egy változót a lokális változók (tetszőlegesen nagy) területén, akárhol is helyezkedik el. Ennek megvalósításához az IJVM ugyanazt a mechanizmust használja, mint a JVM: egy speciális WIDE nevű opcode-t, amelyet egy prefix byte-ként foghatunk fel és amely közvetlenül az ILOAD vagy ISTORE utasítások kódja előtt áll. Ha ez az eset következik be, akkor az ILOAD és az ISTORE utasítások definiciója módosul; 16 bites index követi az utasításkódot a 8 bites helyett, mint a 4-19 (b) ábra mutatja.
A WIDE a szokásos módon dekódolódik, amely a wide 1-hez vezet, ami kezeli a WIDE opcode-t. Bár a "kiszélesítendő" opcode már közvetlenül elérhető az MBR-ben, wide 1 elhozza az opcode utáni első byte-ot, mert a mikroprogram logikája mindig elvárja azt, hogy ott legyen. Ezután egy második többirányú elágazás történik, ezúttal a WIDE-ot követő byte-ot használva elküldésre. Mindazonáltal, mivel a WIDE ILOAD különböző mikrokódot követel mint az ILOAD és a WIDE ISTORE különbözo mikrokódot követel mint az ISTORE stb. ez a második többirányú ág nem használhatja az opcode-t, mint célterület címet, mint ahogy azt a Main 1 teszi.

4-20. ábra Ez az ILOAD és a WIDE ILOAD kezdeti mikroutasítás-sorozata. A címek példaként állnak.

Ehelyett a wide 1 VAGY művelettel összekapcsolja 0x100- t az opcode-dal, mialatt az MPC-be teszi. Eredményképpen a WIDE ILOAD megvalósítása 0x115-ön kezdődik (0x15 helyett), a WIDE ISTORE megvalósítása pedig a 0x136-on kezdődik (0x36 helyett) stb. Ez alapján minden WIDE utasításkód 256-tal (azaz 0x100-zal) magasabb címen kezdődik a vezérlő tárban, magasabban mint a megfelelő reguláris opcode. A mikroutasítások kezdeti sorozata, mind az ILOAD-ra, mind a WIDE ILOAD-ra a 4-20-as ábrán láthatók.
Amint a kód elér a WIDE ILOAD megvalósításához (0x115) a kód a normális ILOAD-tól mindössze annyiban különbözik, hogy az indexet két egymás utáni index byte összefűzésével kell megkonstruálni, ahelyett, hogy csak egy byte-on végeznénk "előjel-kiterjesztést". Az összefűzést és a rákövetkező összeadás műveleteket részletekben kell végrehajtani, először az INDEX BYTE 1-et H-ba másolva és 8 bittel balra tolva. Mivel az index előjel nélküli egészet reprezentál, az MBR-en nulla-feltöltést hajtunk végre, MBRU-t használva. Ezután az index második byte-ját hozzáadjuk (az összeadás itt megfelel a konkatenáció (összefűzés) műveletének, mivel a H alsó byte-ja nulla, így garantálva van, hogy nincs átvitel (carry) a byte-ok között) és az eredmény H-ban marad. Innentől kezdve a művelet pontosan ugyanúgy folytatódhat, mint ha egy standard ILOAD-ról lenne szó. Ahelyett, hogy duplán végrehajtanánk az ILOAD záró utasításait (iload 3-tól iload 5-ig) egyszerűen leágazunk wide-iload 4-től az iload 3-hoz. Figyeljük meg azonban, hogy PC-t kétszer kell megnövelnünk az utasítás végrehajtása során, hogy az a következő utasítás kódjára mutathasson. Az ILOAD megnöveli azt egyszer; a WIDE-ILOAD is megnöveli egyszer.
Ugyanez a szituáció a WIDE-ISTORE-ral, miután az első 4 mikroutasítás

végrehajtódik (wide_istore 1-től 4-ig) a folytatás ugyanaz a sorozat, mint az ISTORE első két utasítása után, így a wide-istore 4 leágazik az istore 3-hoz.
A következő példa, amit tekinteni fogunk az LDC_W utasítás. Ennek a kódja (opcode) két dologban is különbözik az ILOAD-étól. Először is egy 16 bites előjel nélküli relatív címe van (mint az ILOAD wide változatának). Másodszor: a CPP-vel van indexelve LV helyett, mivel a rendeltetése az, hogy a konstans-készletből olvasson a lokális változó keret helyett. (Igazából az LDC_W-nek van egy rövid változata (LDC), de ezt nem vettük be az IJVM-be, mivel a hosszú forma magába foglalja a rövid forma összes lehetséges variációját és csak 3 byte-ot foglal a 2 helyett.)
Az IINC utasítás az egyetlen IJVM utasítás az ISTORE-on kívül, amely egy lokális változót módosítani tud. Ezt úgy teszi meg, hogy két operandusa van, mindegyik 1 byte hosszú: ld. a 4-21 ábrát.

4-21. ábra Az IINC utasításnak két különböző operandusa van.
Az IINC utasítás az INDEX-ben határozza meg az relatív címet a lokális változók memóriaterületének kezdetéhez képest. Kiolvassa a változót, megnöveli az értékét CONST-tal, amelynek értékét az utasítás tartalmazza, és eltárolja az eredményt ugyanarra a helyre. Megjegyezzük, hogy ez az utasítás negatív értékkel is "megnövelhető", azaz a CONST egy 8 bites előjeles egészt reprezentál a -128 - +127 intervallumból. A teljes változatú JVM-ben található egy WIDE verziója is az IINC-nek, ahol minden operandus 2 byte hosszú.
Most érkeztünk el IJVM első elágazási utasításához, a GOTO-hoz. Az egyedüli célja ennek az utasításnak, hogy megváltoztassa a PC értékét, hogy a következő végrehajtandó IJVM utasítás az legyen, amelynek értékét a 16 bites (előjeles) offset és az elágazási utasítás címének összeadásával számítunk ki. A komplikáció itt az, hogy az offset ahhoz képesti relatív eltolást ad meg, amennyi a PC értéke volt az utasítás dekodolásának kezdetén, és nem ahhoz képest, amennyi a PC értéke a 2 byte-os offset betöltése után lett.
Hogy ezt világossá tegyük, 4-22 (a) ábrán azt a helyzetet láthatjuk, amely a Main 1 kezdetén áll fenn. Az utasítás kódja már MBR-ben van, de PC még nem lett megnövelve. A 4-22 (b) ábrán láthatjuk a helyzetet a goto 1 kezdetén. Eddigre a PC már meg lett növelve és az első offset byte már el lett hozva MBR-be. Egy mikroutasítással később a 4-22 (c) ábrán látható helyzethez jutunk, amelyben a régi PC érték, amely az opcode-ra mutat, el lett mentve OPC-be. Erre az értékre azért van szükség, mert az IJVM GOTO utasításának offset-je ehhez képest relatív és nem az akkori PC értékhez. Valójában ez okból volt szükségünk az OPC regiszterre elsősorban.

4-22. ábra A különféle mikroutasítások kezdeti állapota. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

A mikroutasítás goto 2-nél kezdi meg a második offset byte elhozását a 4-22 (d) ábrán látható helyzetet eredményezve a goto 3 kezdetén. Miután az első offset byte 8 bittel balra lett eltolva és H-ban másolódott, elérkezünk goto 4 hez és a 4-22 (e) ábrán levő helyzethez. Ekkor a következő a helyzet: az első offset byte balra lett tolva H-ba, a második offset byte MBR-ben az alap pedig OPC-ben.

Megkonstruálva a 16 bit-es offset-et H-ban és hozzáadva azt az alaphoz megkapjuk a

goto 5-ben PC-be teendő új címet. Vigyázzunk: goto 4-ben MBRU-t használtunk MBR helyett, mivel nem akartunk előjel kiterjesztést a második byte-on. A 16 bites offset valójában a két fél OR művelettel való összekapcsolásával jön létre. Végezetül el kell hoznunk a következő opcode-t mielőtt visszatérnénk Main 1-hez, mivel a kód ott feltételezi, hogy a következő opcode MBR-ben van. Az utolsó időciklus, goto 6 szükséges tehát, mert az adatot a memóriából el kell hozni, hogy időben megjelenjen MBR-ben a Main 1 alatt.
Az IJVM GOTO utasításának offset-jei előjeles 16 bites egészek, amelynek minimális értéke -32768 és maximális értéke +32767. Ez azt jelenti, hogy két ennél egymástól távolabbi cím közötti ugrás nem lehetséges.

***[242-245] javított!

242-245. A mikroarchitektúra szintje

Ezt a tulajdonságot hibaként vagy az IJVM jellemző vonásaként lehet tekintetbe venni. A hibacsoport szerint a JVM nem korlátozhatná az ő programstílusukat. Ez a csoport azt állítja, hogy számos programozó munkája hatalmasat fejlődne, ha megijednének a szerkesztő üzenetétől:

A program túl nagy. Újra kell írni. A szerkesztés sikertelen.

Sajnos ez az üzenet csak akkor jelenik meg, amikor a szakasz meghaladja a 32 Kb-ot, tipikusan kb 50 Java oldalnál.

Most vegyük figyelembe a 3 IJVM feltételes elágazó utasítást: IFLT, IFEQ, IF_ICMPEQ. Mindhárom esetben szükség van az olvasáshoz az új top-of-stack beolvasására és a TOS-ben való tárolásra.

Ennek a három utasításnak a vezérlése hasonló: az operanduszok vannak először betéve a regiszterekbe, ezután az új top-of-stack lesz beolvasva a TOS-ba, végül a tesztelés és az elágazás történik. IFLT-t tekintsük először. A szó, amit tesztelünk, már a TOS-ban van, de miután az IFLT kiolvassa a szót, az új top-of - stack-et be kell olvasni a TOS-ba. Ez a beolvasás iflt1-ben kezdődött.

Ha sikeres, a müködés maradékrésze lényegében ugyanaz, mint a GOTO utasitás kezdetén, és a sorozat egyszerüen a GOTO sorozat közepén folytatódik a goto 2-vel. Ha sikertelen egy rövid számsort (F,F2,F3) át kell ugrani, mielőtt visszatérünk a Main1-be, hogy folytassuk a következő utasításokkal.

A kód ifeq2 és ifeq3-ban ugyanazt a logikát követi, csak Z bitet használ az N helyett. Mindkét esetben a MAL gyűjő feladata, hogy felismerje, hogy a címek T és F speciálisan és, hogy megbizonyosodjon arról, hogy a címek a helyükön vannak a vezérlő tárban csak a baloldali biten térnek el.

A logika az IF_ICMPEQ-ban kb. egyforma az IFLT-vel kivéve azt, hogy itt le kell olvasnunk a második operanduszt is. A második operandusz a H-ban van tárolva, az if_icmpeq3-ban, ahol az új top-of-stcak szó beolvasása elkezdődik. Majd újra el van mentve az OPC-ben az érvényes top-of-stcak szó, és a TOS-ban van elhelyezve. Végül a teszt az if_icmpeq6-ban hasonló az ifeq4-hez.

Most megvizsgáljuk az INVOKEVIRTUAL és az IRETURN végrehajtását, az utasításokat, amelyekhez segítségül hívunk egy eljárást és visszatérünk. INVOKEVIRTUAL 22 mikroutasítás sorából áll és a legösszetettebb IJVM utasítást hajtja végre. (4-12. ábra) Az utasítás 16 bit offset-et használ, hogy megállapítsa a hívott módszer címét. A mi végrehajtásunkban egy egyszerő offset-et használunk a Constant Pool-ban. A Constant Pool-ban ez a cím adja a hívott eljárás helyét. Emlékezni kell arra, hogy minden egyes eljárás első 4 bájtja nem utasítás, hanem két 16 bites mutató. Az első a paraméter szavak számát adja meg (beleértve OBJEF-et -4-12. ábra). A második a helyi, lokális változóterület nagyságát adja meg szavakban. Ezek a mezők keresztül mennek egy 8 bites kapun és össze vannak gyűjtve úgy, mintha 16 bit offset-ek lennének egy utasításon belül.

Aztán a kapcsolódó információnak vissza kell állítania a gépet az előző állapotába -a régi változó terület helyének és a régi PC-nek a címe- ezt azonnal az újonnan létrehozott változó terület felett tároljuk,ami az új verem alatt van. Végül a következő utasítás opcode-ja megérkezik és PC növelődik, mielőtt visszatér a Main-ba, hogy elkezdje a következő utasítást.

IRETURN egy egyszerű utasítás, ami nem tartalmat operanduszt. Ez egyszerűen

a lokális változó terület első szavában tárolt címet használja, hogy elérje a kapcsolódó információt. Azutan helyreállítja az SP, EV, PC -t az előző értékükre és lemásolja a visszatérő értéket a jelenlegi verem tetejéről az eredeti verem tetejére (4-13. ábra).

A micorarchitektura szint design-ja

Mint ahogy eddig is láttuk, minden computer science, a micorarchitektura szint tele van trade-off-okkal. A számítógépeknek nagyon sok jellemzőjük van, mint pl.: gyorsaság, ár, megbízhatóság, gond nélküli használat, energia kívánalom és fizikai méret. Egy trade-off vezeti a fegfontosabb választásokat, a CPU készítőjének készítenie kell egy speed versus cost-ot. Ebben a szekcióban jobban megnézzük ezeket a kimeneteket, azért, hogy megértsük, mit lehet trade-off-olni mi ellen, milyen magas színvonalat lehet elérni, és milyen árban van a hardware és a bonyolultság.

Sebesség elleni ár (speed versus cost)

A sebességet sokféleképpen mérhetjük, de ha adott egy áramkörös technologia és a ISA, három fő tulajdonság segítségével növelhetjük e végrehajtási sebességet:

1. Csökkenteni kell az óraciklusok számát, amelyekre azért van szükség, hogy végrehajtsa az információkat.
2. Leegyszerűsíteni a szervezettséget, hogy az óraciklus rövidebb legyen.
3. Átlapozni az utasítás végrehajtását

Az első kettő nyilvánvaló, de van egy meglepő vált. a lehetőségekben, amelyek kihatnak vagy az óraciklus számára, vagy az óra periódusidejére, vagy leggyakrabban mindkettőre.

Az óraciklusok számának végre kell hajtani beállításokat, amit az elérési út hosszúságaként (path lenght) ismerünk. Néha ezt a elérési út hosszt le lehet rövidíteni azzal, ha hozzáadunk specializált hardware-ket. Pl.: azzal, ha hozzáadunk egy növekedést a PC-hez, nem muszály tovább használni az ALU-t, hogy fejlesszük a PC-t, elhagyhatjuk a ciklust. Azért az árért, amit fizetünk, több hardware-t kapunk. Azonban ez a képesség nem segít annyit, amennyit remélnénk tőle. A legtöbb utasítás, amit a ciklusok használnak, hogy növeljék a PC-t, szintén ciklus, amelyben az olvasott művelet el van végezve. A köv. utasításokat nem lehetett végrehajtani korábban, mert ez a memóriába jövő adattól függött.

Az utasítások ciklusainak a számát csökkentve megérkeznek az információk, amiben több, mint egy új ciklusra van szükség, ahhoz, hogy a PC növekedjen. Ahhoz, hogy felgyorsítsuk az utasítások megérkezését a harmadik technikát -az utasítások végrehajtásának az átlapozását- kell kihasználnunk. Ha szétválasztjuk az áramkört -a 8 bit memória kaput és az MBR és a PC regisztereket- ahhoz, hogy az információk megérkezzenek, az a leghatásosabb, ha az egység független az adat fő elérési útjától. Ebben az esetben egyedül el tudja hozni a köv. opcod-ot vagy operand-ot, talán még nem egyidőben működve végrehajtja a CPU megmaradt részét, és egy vagy több utasítást hoz előre. A sok utasítás végrehajtásának az egyik legidőigényesebb fázisa az, amikor két bite offset-te kell elhozni, megfelelően meghosszabbítani ls felhalmozni a H regiszterben és ujra felhasználni, pl: egy elágazásban a PC+- N bite-jában. Egy lehetséges megoldás -a memória kaput 16 bitre növeljük- megbonyolítja a mőködést, mert a memória 32 bit széles. A 16 bit-nek span words határvonalakra van szüksége, tehát egy egyszerűen leolvasott 32 bit nem feltétlenül fogja elhozni mindkét szükséges bite-ot.

Az utasítások végrehajtásának az átlapozása messze a legérdekesebb és ez biztosítja a legjobb lehetőséget arra, hogy megnöveljük a sebességet, Az utasítások elhozatalának és végrehajtásának egyszerű átlapozása nagyon hatásos. A kifinomultabb technikák előrébb jutnak akár hogyis lapozzuk át az utasítások végrehajtását. Valójában ez az ötlet a modern computer design szivében van.

A sebesség a kép fele; az ár a másik fele. Az árat sokféleképpen mérhetjük, de az ár pontos meghat. problematikus. Néhány mérés olyan egyszerű, mint az alkotórészek számának a megszámolása. Ez akkor volt igaz, amikor a processzorok különálló alkatrészekből voltak építve, amelyeket beszereztek és összegyűjöttek. Ma az egész processzor egy chip-ben van, de a nagyobb összetettebb chip-ek drágábbak, mint a kissebbek, és az egyszerűbbek. Az önálló alkatrészeket pl.: tranzisztorok, gates-ek vagy működési egységeket bele lehet számolni, de gyakran a számítás nem olyan fontos, mint a területnagyság, amit megkíván a rendezett áramkör. Minél nagyobb területet kíván meg a funkció, annál nagyobb a chip. E chipnek az előállítási ára gyorsabban növekszik, mint a tér. A leggyorsabban tanulmányozott áramkörök egyike a bináris ... . Ezer féle design volt és a leggyorsabb gyorsabb, mint a leglassúbb, és sokkal összetettebb is. A rendszer tervezőjének el kell döntenie, hogy a nagyobb gyorsaság megérje a real-estate-et.

A cím nem az egyetlen lehetőségekkel ellátott alkotóelem. Majdnem mindegyik alkotóelemet a rendszerben gyorsabbra és lassúbbra lehet tervezni árkülönbséggel. A készítő számáta ez a kihívás, hogy felismerje azokat az alkotóelemeket a rendszerben amelyek a legjobban fejleszthetik a rendszert azzal, hogy felgyorsulnak. Sok egyedülálló alkotóelemet lehet helyettesíteni sokkal gyorsabb alkotóelemekkel, amelyeknek nincs, vagy alig van hatása a sebességre. A kulcstényezők egyike azt határozza meg,hogy amilyen gyorsan az óra megy, olyan gyorsan kell elvégezni a munkát mindegyik óraciklusban. Nyilvánvalóan, minél több munkát elvégez annal hosszabb lesz az óraciklus. Ez nem annyira egyszerű term. mert a hardware nagyon jó abban, hogy párhuzamosan végezze a munkát, tehát valójában folytatásos müködés az, amit végre kell hajtani sorozatosan egy egyszerű óraciklusban, ami meghatározza, hogy az óraciklusnak milyen hosszúnak kell lennie.

Egy szemlélet van, amit kontrolálni lehet, az pedig a dekódolás nagysága, amit meg kell formálni. Láthattuk a 4-6. ábrán, hogy mialatt a 9 regiszter bármelyikét le lehet olvasni ALU-ba a B bus-ból, nekünk csak 4 bit-re volt szükségünk a microinstruction szóban, hogy meghatározzunk, melyik regisre lett kiválasztva. Sajnos ezek a mentések költségesek. A dekódolt áramkör késlelteti az áramkör elérési útját. Ez azt jeleneti, hogy a B bus-ban majd kapni fog egy parancsot, és később meg fogja kapni az adatot a Bus-on.

A határzuhatag az ALU-val kapja majd meg a bemenetet, és az eredményt is később fogja előhozni. Végül az eredmény rendelkezésre áll a C Bus-ban ahhoz, hogy leírják a regisztert egy kicsit később. Amióta ez a kérés gyakran a factor, ez meghatározza, milyen hosszúnak kell lennie az óraciklusnak, ez azt is jelenti, hogy az óra nem mehet olyan gyorsan, és az egész számítógépnek lassabban kell dolgoznia. Van egy trade -...- a sebesség és az ár között. Csökkentve a controll store-t 5 bit/szóként az óra lelassításának az ára jön. A design készítőjének muszály elszámolnia a design targyait, amikor eldönti, hogy melyik a helyes választás. Egy magas szívvonalú véghezvitelhez dekódert használni nem a legjobb ötlet.

Lecsökkenteni a végrehajtási elérési út hosszát

A Mic-1 arra volt tervezve, hogy közepesen egyszerű és gyors legyeb, bár kétség kívül nagyfeszültség van a kettő goal között.

***[246-249]

246-249. oldal

A mikroarchitektúra szintje

A MIC-1 CPU-nak szintén szüksége van valamennyi hardware-re: tíz regiszterre, az ALU-ra (a 3-19. ábrán) 32-szer megsokszorozva, egy shifterre (kapcsolókarra), egy dekóderre és egy vezérlő készletre. Az egész rendszert fel lehetne építeni alig 5000 tranzisztorból és mindabból, amire a vezérlő készletnek (a ROM-nak) és a központi tárnak (a RAM-nak) szüksége van.
Látva, hogy az IJVM-et hogyan lehet egyszerűen mikrokóddal felszerelni, kevés hardware használatával, most itt az ideje, hogy figyeljünk a gyorsabb végrehajtás lehetőségére. Először meg fogjuk nézni, hogyan lehet lecsökkenteni a mikroutasítások számát az ISA utasítások által (csökkentve a végrehajtási szakasz hosszát). Ezután más megközelítést veszünk figyelembe.

Az értelmező ciklus egyesítése a mikrokóddal

A Mic-1-ben a fő ciklus egy mikroutasításból áll, aminek teljesülnie kell minden IJVM utasítás elején. Néhány esetben lehetséges, hogy ezt túlságosan átfedi az előző utasítás. Valójában ez már részlegesen végre van hajtva. Figyeljük meg, hogy amikor Main1 végrehajtotta az opkód értelmezését, akkor az már az MBR-ben van. Az opkód vagy azért van ott, mert az előző fő ciklus által be lett hozva (ha az előző utasításnak nem volt operandja), vagy az előző utasítás végrehajtása alatt lett behozva.
Az a fogalom, hogy az utasítások kezdetén átfedések találhatók, még tovább vihető, és valójában a fő ciklus lecsökkenthető a nullára. Ez a következőképpen fordulhat elő. Figyeljük meg minden mikroutasítás elrendezését, hogy milyen ágazással végződik Main1-ben. A fő ciklus mikroutasítás mindegyik helyen oda van illesztve a sorozat végére (inkább, mint a következő sorozat elejére), a multiway ággal, ami most sok helyet másol (de mindig ugyanazzal a céllal). Néhány esetben a Main1 mikroutasítást egyesíteni lehet az előző mikroutasításokkal, mivel az utasítások nincsenek mindig teljesen felhasználva.
A 4-23. ábrán a dinamikus utasítássorozatokban mutatkoznak a POP utasítások. A fő ciklus előfordul minden utasítás előtt és után, a számokban mi csak az előfordulást mutatjuk a POP utasítások után. Vegyük figyelembe, hogy ezeknek az utasításoknak a végrehajtása négyórás ciklust vesz igénybe: három óra a POP-nak megadott különleges mikroutasítások számára és egy óra a fő ciklus számára.

4-23.ábra: Új mikroprogram sorozat a POP végrehajtása számára:

CÍMKE	MŰKÖDÉS	MAGYARÁZAT
pop1	MAR=SP=SP-1;rd	a stack-en levő next-to-top olvasása
pop2		várj az új TOS-ra, hogy le legyenek olvasva a memóriából
pop3	TOS=MDR; goto Main1	másold az új szót a TOS-ba
Main1	PC=PC+1; fetch; goto (MBR)	MBR holds opcode; kapja a következő byte-ot; dispatch

A 4-24. ábrán a sorozat le lett csökkentve három utasításra azzal, hogy egyesítették a fő ciklus utasításait, előnyökhöz juttatva az óra ciklust, amikor az ALU nincs használat alatt a pop2-ben, azért, hogy elmentse a ciklust és újra a Main1-be kerüljön. Figyelembe kell venni, hogy ezeknek az elágazássorozatoknak a vége közvetlenül a speciális kódokhoz "megy" a sorozati utasításért, tehát csak három ciklus szükséges az összegzéshez. Ez a kis trükk lecsökkenti a következő mikroutasításnak a végrehajtási idejét egy ciklussal, tehát például, a rákövetkező IADD négy ciklusról háromra megy. Ennek következtében egyértelmű az órának a felgyorsítása 250 Mhz-ről 333 Mhz-re szabadon.

4-24.ábra: Új mikroprogram sorozat a POP végrehajtása számára:

CÍMKE	MŰKÖDÉS	MAGYARÁZAT
pop1	MAR=SP=SP-1;rd	a stack-on levő next-to-top olvasása
Main1.pop	PC=PC+1;fetch	MBR holds opcode; fetch next byte
pop3	TOS=MDR; goto (MBR)	Másold az új szót a TOS-ba; dispatch on opcode

A POP utasítás tökéletesen megfelel erre az eljárasmódra, mert van egy holt ciklusa, középen, amit az ALU nem használ. A fő ciklus használja az ALU-t. Az egyel lecsökkentett utasítások hosszúsága egy utasításban szükségessé teszi, hogy találjunk egy ciklust az utasításban, ahol az ALU-t nem használjuk. A holt ciklusok nem mindennaposak, de előfordulnak, tehát egyesítik Main1-et mindegyik mikroutasítás sorozatának a végén, amit érdemes elvégezni. Ez csupán a vezérlő készletbe kerül. Ugyanakkor már ismerjük az első technikát arra, hogy lecsökkentsük az elérési út hosszúságát: Egyesítsük az értelmező ciklust mindegyik mikrokód sorozat végével.

A három-busz architektúra

Mit tudunk még csinálni, hogy lecsökkentsük a végrehajtási szakasz hosszát? Egy másik könnyű lehetőség, hogy két teljes bemeneti busz az ALU-ba, egy A és egy B, összesen három buszt ad. Mindegyik regiszternek (vagy legalábbis lagtöbbjüknek) szabad bejárással kellene rendelkeznie mindegyik bemeneti buszba. Az az előnye annak, ha két bemeneti buszunk van, hogy lehetségessé válik, hogy hozzáadjunk bármilyen regisztert más regiszterekhez egy ciklusban. Ahhoz, hogy az érték sajátosságait lássuk, figyelembe kell vennünk a Mic1 ILOAD végrehajtását (4-25. ábra).

4-25.ábra: Mic-1 kód az ILOAD végrehajtása számára:

CÍMKE	MŰKÖDÉS	MAGYARÁZAT
iload1	H=LV	MBR tartalmaz indexet; LV másolása H-ra
iload2	MAR=MBRU+H;rd	MAR=lokális változó címe
iload3	MAR=SP=SP+1	SP a verem új topjára mutat; elkészíti az írást

iload4 PC=PC+1; fetch; wr Inc PC; kapja a következő opkóde; írott verem topja
iload5 TOS=MDR; goto Main1 Update TOS
Main1 PC=PC+1; fetch; goto (MBR) MBR holds opcode; kapja a következő byte-ot; dispatch

Az iload1 LV másolva van H-ra. Azért másolták át, hogy így hozzá lehessen H-t adni MBRU-hoz az iload2-ben. A mi eredeti két buszos tervünkben nincs arra lehetőség, hogy hozzáadjunk két tetszőleges regisztert, tehát egyiküket először H-ra kell másolni. Az új három busz tervvel meg tudunk menteni egy ciklust. Hozzáadtuk az értelmező ciklust az ILOAD-hoz, de ez nem növelte és nem is csökkentette az ILOAD végrehajtási idejét hat ciklusról öt ciklusra. Most már megvan a második technikánk arra, hogynöveljük az elérési út hosszúságát: Két busz tervből három busz terv lesz.

4-26.ábra: Három busz kód az ILOAD végrehajtása számára:

CÍMKE	MŰKÖDÉS	MAGYARÁZAT
iload1	MAR=MBRU+LV;rd	MAR=lokális változó címe
iload2	MAR=SP=SP+1	PC a legújabb fenti veremre mutat; elkészíti az írást
iload3	PC=PC+1;fetch;wr	Inc PC;kapja a következő opkódot; írja a legfelső verembe
iload4	TOS=MDR	Update TOS
iload5	PC=PC+1;fetch;goto (MBR)	MBR-ben már van opkód; elhozza az index byte-ot

A Fetch Unit utasítás

Mindkét technikát megéri használni, de ahhoz, hogy nagyfokú fejlődést érjünk el, sokkal mélyebbre kell jutnunk. Lépjünk vissza és nézzük meg minden utasításnak a mindennapi oldalát: az utasításos mezők elhozása és kiterjesztése. Figyelembe kell vennünk, hogy minden utasításnál a következő működés fordulhat elő:
1. A PC áthalad az ALU-n és növekszik.
2. A PC-t arra használják, hogy a következő byte-okat az utasításos folyamatba hozza.
3. Az operandokat a memóriából lehet olvasni.
4. Az operandok a memóriába vannak írva.
5. Az ALU végzi a számítást és az eredményeket tárolja.

Ha egy utasításnak van egy újabb mezője, akkor minden mezőt muszáj határozottan elhozni, egy byte-ot egy alkalommal, és összegyűjteni, mielőtt használjuk. Egy mező elhozatala és összegyűjtése korlátozza az ALU-t, legalább egy ciklusban byte-onként, abban, hogy növelje a PC-t, és azután újra, hogy összegyűjtse az eredményül kapott tárgymutatót vagy az offset-et. Az ALU-t majdnem minden ciklusnál használjuk, különböző műveletekre, amelyeknek az utasítások elhozatalához és a mezők összegyűjtéséhez van köze egy utasításon belül, azonkívül az utasítás valódi "munkájához". A fő ciklus túlságos átfedése miatt szükséges felszabadítani az ALU-t néhány feladat alól. Ez úgy is elérhető, ha bevezetjük a második ALU-t, bár egy teljes

ALU nem szükséges az aktiváláshoz. Figyelembe kell venni, hogy számos esetben az ALU-t egyszerűen mint mezőt használjuk arra, hogy átmásoljon egy értéket egy regiszterből a másikba. Ezek a ciklusok kiküszöbölhetőek lehetnének azzal, ha bevezetnénk egy újabb adat elérési utat, amely nem menne keresztül az ALU-n. Egy kis haszon is származhat, például, ha TOS-ból és MDR-be, készítünk egy elérési utat, vagy MDR-ből TOS-ba , amiután a stack fő szava gyakran át van másolva e között a két regiszter között.
A Mic1-ben a betöltés nagy részét el lehet távolítani az ALU-ból úgy, hogy készítünk egy független egységet, hogy elhozza és feldolgozza az utasításokat. Ez az egység, amit IFU-nak (Instruction Fetch Unit) önállóan tudja a PC-t növelni és elhozni a byte-okkal a byte-"özön"-ből, mielőtt szükség van rájuk. Ez az egység csak egy növelést követel meg, egy körforgás sokkal egyszerűbb, mint egy teljes átadás.Továbbvisszük ezt az ötletet, az IFU még össze tud gyűjteni 8 és 16 bit operandot, tehát ezek készen állnak a közvetlen használatra bármikor, amikor szükséges. Két módja van annak, hogy ez tökéletes legyen:

1. Az IFU minden operandot tud értelmezni, meghatározva, hogy hány további mező kell az elhozatalhoz, és összegyűjti őket egy regiszterbe, készen állva a használatra a fő végrehajtó egység által.
2. Az IFU-nak haszna származhat az utasítások jellegének "folyamából" és mindig rendelkezésre állnak a következő 8 és 16 bites darabok, akár van értelmük, akár nincs. A fő végrehajtási egység akármit "kérhet", amire csak szüksége van.

Bemutatjuk a második terv alapismereteit. Egy egyedüli 8 bit MBR helyett most két MBR van: a 8 bit MBR1 és a 16bit MBR2. Az IFU nyomokat hagy a legkorábbi byte-okban, felhasználva a fő végrehajtási egységet.

***[250-253]
Nem érkezett meg. Herpai Györgyi:

***[254-257]

{Kiss Róbert   254.-257.oldal}
254.

Elnevezés	Műveletek	Magyarázatok
Nop1	goto(MBR)	Elágazás a következő utasításhoz
Iadd1	MAR = SP = SP-1;rd	Beolvasni a tetővel szomszédos szót a verembe
Iadd2	H = TOS	H = a verem tető
Iadd3	MDR = TOS = MDR+H;wr;goto(MBR1)	A veremhez adni két szót ,beírni az új veremtetőt
Isub1	MAR = SP = SP-1;rd	Beírni a tetővel szomszédos szót a verembe
Isub2	H = TOS	H = a verem tető
Isub3	MDR = TOS = MDR-H;wr;goto(MBR1)	A TOS-t kivonni a lehívott TOS-1-ből
Iand1	MAR = SP = SP-1;rd	Beírni a tetővel szomszédos szót a verembe
Iand2	H = TOS	H = a verem tető
Iand3	MDR = TOS = MDR AND H;wr;goto(MBR1)	A lehívott TOS-t AND utasítással összekötni a TOS-sal
Ior1	MAR = SP = SP-1;rd	Beírni a tetővel szomszédos szót a verembe
Ior2	H = TOS	H = a verem tető
Ior3	MDR = TOS = MDR OR H;wr;goto(MBR1)	A lehívott TOS-t OR utasítással összekötni a TOS-sal
Dup1	MAR = SP = SP+1	Növelni SP-t ,bemásolni MAR-ba
Dup2	MDR = TOS;wr;goto(MBR1)	Beírni az új verem-szót
Pop1	MAR = SP = SP-1;rd	Beírni a tetővel szomszédos szót a verembe
Pop2		Várni a beírásra
Pop3	TOS = MDR;goto(MBR1)	Bemásolni az új szót a TOS-ba
Swap1	MAR = SP-1;rd	Kiolvasni a második szót a veremből
Swap2	MAR = SP	Előkészülni új második szó írására
Swap3	H = MDR;wr	Kimenteni az új TOS-t ,második szótbeírni a verembe
Swap4	MDR = TOS	A régi TOS-t bemásolni MDR-be
Swap5	MAR = SP-1;wr	A régi TOS-t beírni a verem második helyére
Swap6	TOS = H;goto(MBR1)	Tos-t felül írni
Bipush1	SP = MAR = SP+1	MAR-t beállítani az új verem tető írására
Bipush2	MDR = TOS = MBR1;wr;goto(MBR1)	A vermet felülírni a TOS-ban és tárolni
Iload1	MAR = LV+BR1U;rd	LV + index mozgatása MAR-ra ,operandus beolvasása
Iload2	MAR = SP =SP+1	SP növelése ,az új SP mozgatása MAR-ra
Iload3	TOS = MDR;wr;goto(MBR1)	A vermet felülírni a TOS-ban és tárolni
Istore1	MAR = LV+MBR1U	MAR-t beállítani LV + index-re
Istore2	MDR = TOS;wr	TOS-t másolni tároláshoz
Istore3	MAR = SP = SP-1;rd	SP-t csökkenteni ,beolvasni az új TOS-t

Istore4		Várni a beolvasásra
Istore5	TOS = MDR;goto(MBR1)	TOS-t felülírni
Wide1	goto(MBR1 OR 0x100)	A következő cím 0x100 or utasítás gépi kóddal
Wide_iload1	MAR = LV+MBR2U;rd;goto iload2	Azonos iload1-gyel ,a 2-byte-os index alkalmazásával
Wide_istore1	MAR = LV+MBR2U;goto istore2	Azonos istore1-gyel ,a 2-byte-os index alkalmazásával
Ldc_w1	MAR = CPP+MBR2U;rd;goto iload2	Ugyanaz ,mint a széles_iload1 ,a CPP leindexelésével
Iinc1	MAR = LV+MBR1U;rd	MAR beállítása LV+indexre beolvasásra
Iinc2	H = MBR1	H beállítása konstansra
Iinc3	MDR = MDR+H;wr;goto(MBR1)	Növelni a konstanssal és felülírni
Goto1	H = PC-1	PC-t bemásolni H-ra
Goto2	PC = H+MBR2	Hozzáadni eltolást és PC-t felülírni
Goto3		Várni az utasítást lehívó egységre ,hogy lehívja a gépiutasítást
Goto4	goto(MBR1)	Elküldeni a következő utasításhoz
Iflt1	MAR = SP = SP-1;rd	Beolvasni a tetővel szomszédos szót a verembe
Iflt2	OPC = TOS	A TOS-t kimenteni átmenetileg az OPC-ben
Iflt3	TOS = MDR	Új verem tetőt helyezni a TOS-ba
Iflt4	N = OPC;if (N) goto T;else goto F	Elágazás az N biten

4-30.ábra : Mikroprogram a Mic-2 integrált áramkörhöz(2/1)

255

Elnevezés	Műveletek	Magyarázatok
Ifeq1	MAR = SP = SP-1;rd	Beírni atetővel szomszédos szót a verembe
Ifeq2	OPC = TOS	A TOS-t kimenteni átmenetileg az OPC-ben
Ifeq3	TOS = MDR	Új verem tetőt helyezni a TOS-ba
Ifeq4	Z = OPC;if (Z) goto T;else goto F	Elágazás a Zbiten
If_icmpeq1	MAR = SP = SP-1;rd	Beírni a tetővel szomszédos szót a verembe
If_icmpeq2	MAR = SP = SP-1	MAR beállítása az új verem tető beírására
If_icmpeq3	H = MDR;rd	Bemásolni H-ba a második verem szót
If_icmpeq4	OPC = TOS	A TOS-t kimenteni átmenetileg az OPC-ben
If_icmpeq5	TOS = MDR	Új verem tetőt helyezni a TOS-ba
If_icmpeq6	Z = H-OPC;if (Z) goto T;else goto F	Ha a két tető szó azonos ,goto T ,egyébként goto F
T	H = PC-1;goto goto2	Ugyanaz ,mint goto1
F	H = MBR2	MBR2-ben byte-ok érintése érvénytelenítésre
F2	goto(MBR1)	
Invokevirtual1	MAR = CPP+MBR2U;rd	MAR-ban címeket adni a művelet-mutatóknak
Invokevirtual2	OPC = PC	OPC-ben kimenteni RETURN PC-t

Invokevirtual3	PC = MDR	PC-t a műveletkód első byte-jára állítani
Invokevirtual4	TOS = SP-MBR2U	TOS = az OBJREF-1 címe
Invokevirtual5	TOS = MAR = H = TOS+1	TOS = az OBJREF címe
Invokevirtual6	MDR = SP+MBR2U+1;wr	OBJREF-t felülírni a kapcsolat-mutatóval
Invokevirtual7	MAR = SP = MDR	SP és MAR beállítása olyan helyzetbe ,hogy megtartsuk a régi PC-t
Invokevirtual8	MDR = OPC;wr	Előkészülni a régi PC kimentésére
Invokevirtual9	MAR = SP = SP+1	Megnövelni SP-t olyan helyzetbe ,hogy megtartsuk a régi LV-t
Invokevirtual10	MDR = LV;wr	A régi LV kimentése
Invokevirtual11	LV = TOS;goto(MBR1)	LV beállítása a 0-dik paraméterre
Ireturn1	MAR = SP = LV;rd	SP és MAR átállítása a kapcsolati mutató beolvasására
Ireturn2		Várni a kapcsolati mutatóra
Ireturn3	LV = MAR = MDR;rd	LV és MAR beállítása akapcsolati mutatóra ,a régi PC beolvasása
Ireturn4	MAR = LV+1	MAR beállítása a régi LV-re ,régi LV beolvasása
Ireturn5	PC = MDR;rd	PC visszaállítása
Ireturn6	MAR = SP	
Ireturn7	LV = MDR	LV visszaállítása
Ireturn8	MDR = TOS;wr;goto(MBR1)	A visszatérítési érték kimentése az eredeti verem tetőn

4-30.ábra : Mikroprogram a Mic-2 integrált áramkörhöz(2/2)

az ALU és léptető vezérli őket és ezt követi az eredmények visszaírása a regiszterekbe
Az utasítást lehívó egység /IFU/ kivételével párhuzamosság nincs jelen.
Párhuzamosság betoldása reális lehetőség

Amint korábban említettük ,az óra ciklust azon idő korlátozza ,amely a jelek adatpályán át történő
terjedéséhez szükséges. A 4-3 ábra mutatja a késleltetés letörését a különböző komponensek által az egyes
ciklusok alatt. A tényleges adatpálya ciklusnak három fő komponense van :

1. Azon idő ,amely a kiválasztott regisztereknek az A és B buszokba vezérléséhez szükséges.
2. Azon idő ,amely az ALU és a léptető munkájának elvégzéséhez szükséges
3. Azon idő ,amely az eredményeknek a regiszterekbe történő visszajuttatásához és tárolásához szükséges.

A 4-31 ábrán látjuk az új három buszos szerkezetet ,beleértve az IFU-t ,három kiegészítő átmeneti tárolóval
(regiszterek), egy közbeiktatva az egyes buszok közepére.

256-257

Minden egyes cikluson írva vannak az átmeneti tárolók. Valójában a regiszterek az adatpályát három különálló részre osztják ,amelyek egymástól függetlenül működhetnek. Erre úgy fogunk hivatkozni mint a Mic-3 ,vagy a "futószalag technikás" modell.

{ÁBRA:}

A fő memóriától és -hoz A memóriát vezérlő regiszterek

Utasítást lehívó egység (IFU)

Vezérlő jelek : engedélyez a B buszba

a C buszt a regiszterhez írja

C átmeneti tároló A átmeneti tároló B átmeneti tároló

ALU = aritmetikai és logikai egység

Shifter = léptető

4-31. ábra : A Mic-3 integrált áramkörben alkalmazott három buszos adatpálya

Ezen kiegészítő regiszterek hogyan tudnak esetleg segíteni? Nos ,az adatpálya használatához három óra ciklusra van szükség ,egy az A és B átmeneti tárolók feltöltéséhez ,egy a futtatáshoz ,az ALU-hoz és léptetőhöz és a C átmeneti tároló feltöltéséhez és egy a C átmeneti tárolónak a regiszterbe történő visszatárolásához.

Őrültek vagyunk?(Megjegyzés: Nem.) Az átmeneti tárolók közbeiktatásának lényege kettős:

1. Felgyorsítjuk az órát ,mivel a maximális késleltetés most rövidebb.
2. Minden ciklus alatt használjuk az adatpálya minden részét.

Az adatpálya három részre osztásával csökkent a maximális késleltetés ,azzal az eredménnyel ,hogy az óra frekvencia magasabb lett. Tételezzük fel hogy az adatpálya ciklust három időintervallumra osztva ,ezek mindegyike az eredetinél kb. 1/3-dal rövidebb ,így megháromszorozhatjuk az óra sebességét.(Ez nem teljesen reális ,mivel az adatpályába kettővel több regisztert vittünk be ,de első közelítésben elfogadható)

Mivel feltételeztük ,hogy valamennyi memória beolvasását és beírását ki lehet elégíteni az első szint chache-tárából és ez ugyanazon anyagból készült mint a regiszterek ,feltehetjük továbbá ,hogy a memória művelet egy ciklust foglal el. Bár a gyakorlatban ezt nem könnyű elérni.

A második pont inkább az átbocsátással ,mint az egyes utasításokkal foglalkozik A Mic-2-ben az egyes óra ciklusok első és harmadik részében az ALU inaktív. Az adatpályát három részre osztva az ALU-t minden egyes ciklusban használhatjuk ,háromszor annyi munkát nyerhetünk a gépből.

Lássuk most ,hogyan működik a Mic-3 adatpályája. Indítás előtt az átmeneti tárolók kezeléséhez egy jelzés-rendszerre van szükségünk. Nyilvánvaló ,hogy az átmeneti tárolókat A,B,C-nek

nevezzük és ezeket regiszterként
kezeljük ,szem előtt tartva az adatpálya kényszereit. A 4-32 ábra egy kód sorrend példát mutat ,a SWAP megvalósítását a Mic-2-re.

Swap1	MAR = SP-1;rd	A veremből olvasni a második szót ,a MAR-t SP-re állítani
Swap2	MAR = SP	Előkészíteni a második szó beírását
Swap3	H = MDR; wr	Elmenteni az új TOS-t ,a második szót a verembe írni
Swap4	MDR = TOS	A régi TOS-t másolni az MDR-be
Swap5	MAR = SP-1;wr	A régi TOS-t a verem második helyére írni
Swap6	TOS = H;goto(MBR1)	A TOS-t felülírni

4-32.ábra: A Mic-2 integrált áramkör kódja a memóriák közötti átvitelhez.

Valósítsuk meg most újra ezt a sorrendet a Mic-3-on. Emlékezzünk arra ,hogy az adatpálya most három ciklus működését igényli ,egy tölti A-t és B-t ,egy végrehajtja a műveletet és feltölti C-t és ,egy visszaírja az eredményeket a regiszterekbe. Ezen részek mindegyikét mikrolépésnek fogjuk nevezni.

A 4-33 ábra mutatja a SWAP megvalósítást a Mic-3-ra. Az 1 ciklusban indulunk a swap1-gyel SP-t B-be másolva. Nem számít ,hogy A-ban mi történik ,mivel 1 kivonása B-ből ENA-t érvényteleníti(lásd 4-2 ábrát).

***[258-261]

258-261 oldal
Borbély Gábor (KPM I.)

olyan hozzárendeléseket mutatnak, amiket a gép nem használ. A 2-es ciklusban elvégezzük a kivonást. A 3-as ciklusban az eredmény a MAR-ban tárolódik és a 3-as ciklus végén elkezdődik az olvasási művelet (miután a MAR-t eltároltuk). Mivel a memóriaolvasások most egy ciklust igényelnek, ez az egy csak a 4-es ciklus végén végez, ezt jelöli az MDR-hez való

	Csere1	Csere2	Csere3	Csere4	Csere5	Csere6
Cikl	MAR=SP-1;rd	MAR=SP	H=MDR;wr	MDR=TOS	MAR=SP-1;wr	TOS=H;goto (MBR1)
1	B=SP					
2	C=B-1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B-1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

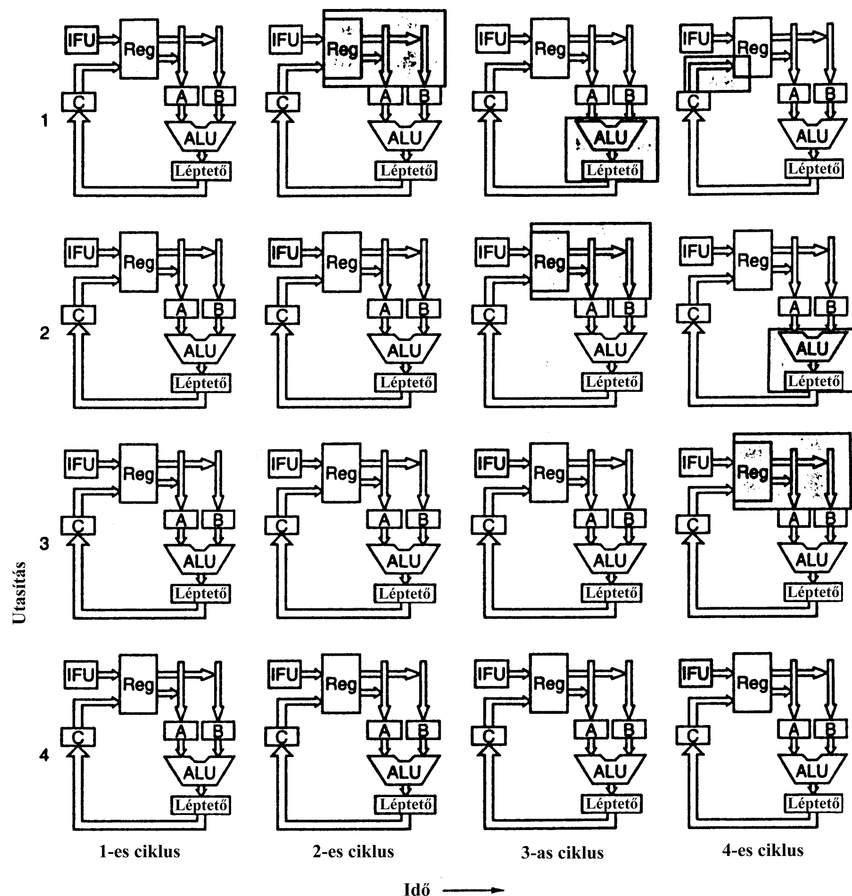
4-33. ábra: CSERE megvalósítása a Mic-3-on

hozzárendelés a 4-es ciklusban. Az MDR-ben lévő értéket nem lehet az 5-ös ciklus előtt olvasni.

Most menjünk vissza a 2-es ciklushoz. Most elkezdhetjük csere2-t mikrolépésekké bontani, és ezeket el is indíthatjuk. A 2-es ciklusban SP-t B-be másolhatjuk, aztán keresztülfuttathatjuk az ALU-n a 3-as ciklusban és végül tárolhatjuk a MAR-ba a 4-es ciklusban. Eddig minden rendben. Tisztában kell lennünk azzal, hogy ha ilyen mértékben tudunk haladni, hogy minden ciklusban új mikroutasítást indítunk, sikerül megtripláznunk a gép sebességét. Ez az előny onnan jön, hogy minden óraciklusra új mikroutasítást adhatunk ki, és a Mic-3-nak háromszor annyi óraciklusa van egy másodperc alatt, mint a Mic-2-nek. Tulajdonképpen egy csővezetékes CPU-t építettünk (pipelined CPU).

Sajnos akadályba ütköztünk a 3-as ciklusban. El szeretnénk kezdeni dolgozni csere3-mon, de az első, amit az tesz, hogy keresztülfuttatja az MDR-t az ALU-n, és az MDR csak az 5-ös ciklus elejére lesz elérhető a memóriából. Azt a helyzetet, amikor egy mikrolépés nem tud elindulni, mert vár arra az eredményre, amit az előző

mikrolépés még nem állított elő, **valós függésnek (true dependence)** vagy **RAW függésnek (RAW dependence)** nevezzük. A függéseket gyakran **kockázatnak (hazard)** hívják. A RAW az angol Read After Write (olvasás írás után) rövidítése, jelzi, hogy egy mikrolépés egy olyan regisztert akar olvasni, ami még nincs megírva. Az egyetlen ésszerű dolog, amit tehetünk, az az, hogy késleltetjük csere3 elindítását amíg az MDR igénybe vehető nem lesz, az 5-ös ciklusban. Egy szükséges értékre várás miatti leállást **várakozásnak (stalling)** hívjuk. Ezután folytathatjuk, hogy minden ciklusban mikroutasításokat kezdünk, mivel már nincsenek függések, bár a csere6-nak éppen csak sikerül elkerülnie, hiszen H-t olvassa a ciklusban, miután azt csere3 beírta. Ha csere5 próbálta volna H-t olvasni, várakoznia kellett volna egy ciklust.



4-34. ábra: A csővezeték működésének grafikus illusztrációja

Annak ellenére, hogy a Mic-3 program több ciklust igényel, mint a Mic-2 program, mégis gyorsabban fut. Ha a Mic-3 ciklusidejét ΔT nsec jelöli, akkor Mic-3-nak 11ΔT nsec-ra van szüksége, hogy végrehajtsa a CSERÉ-t. Ellenben a Mic-2-nek 6 ciklusra van szüksége, mindegyik 3ΔT-s, az összesen 18ΔT. A csővezeték gyorsított a gépen, még akkor is, ha egyszer várakoznunk kellett, hogy elkerüljünk egy függést.

A csővezeték egy kulcsfontosságú technika az összes modern CPU-ban, ezért fontos jól megérteni. A 4-34. ábrán a 4-31. ábra adatpályáját látjuk, grafikusan csővezetéknek ábrázolva. Az első oszlop mutatja be, mi történik az 1-es ciklus során, a második a 2-es ciklust, és így tovább (feltételezve, hogy nincs várakozás). Az árnyékolt terület az 1-es ciklusban az 1-es utasításnál azt jelzi, hogy az IFU el van foglalva az 1-es utasítás lehívásával. Egy órajellel később, a 2-es ciklus során az 1-es utasításhoz szükséges regisztereket betöltjük az A és B tárolókba, míg egyidőben az IFU el van foglalva a 2-es utasítás lehívásával, amit megint a 2-es ciklusbeli két árnyékolt téglalap mutat.

A 3-as ciklus során az 1-es utasítás az ALU-t és az léptetőt használja műveletének elvégzéséhez, az A és B tároló a 2-es utasítás számára feltöltésre kerül, és lehívjuk a 3-as utasítást. Végül a 4-es ciklus során egyszerre négy utasításon dolgozunk. Az 1-es utasítás eredményei eltárolásra kerülnek, a 2-es utasítás számára végrehajtódik az ALU munkája, az A és B tárolókat feltöltjük a 3-as utasítás számára, és lehívjuk a 4-es utasítást.

Ha bemutattuk volna az 5-ös és a rákövetkező ciklusokat, a mintázat ugyanolyan lett volna, mint a 4-es ciklusban: az adatpálya mind a 4 része, amelyek képesek egymástól függetlenül dolgozni, így is tennének. Ez a tervezés egy 4-szintes csővezetéket képvisel; utasításlehívás, operandus elérés, ALU műveletek és regiszterekbe visszaírás számára vannak szintek. Ez hasonló a 2-4(a). ábrán látható csővezetékhez, kivéve, hogy itt nincs a dekódoló szint. A legfontosabb azt megérteni, hogy noha egy különálló utasítás elvégzéséhez négy óraciklus szükséges, egy új utasítás indul, és egy régi utasítás végez.

A 4-34. ábrát máshogy is nézhetjük, úgy, hogy vízszintesen követünk minden egyes utasítást az oldalon. Az 1-es utasítást nézve, az 1-es ciklusban az IFU dolgozik rajta. A 2-es

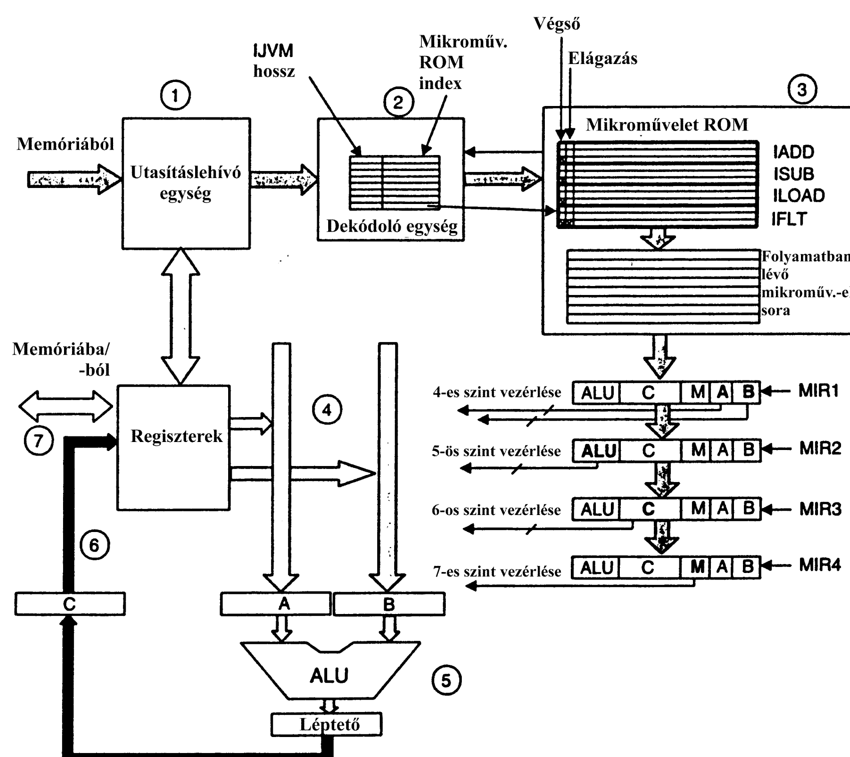
ciklusban a regiszterei az A és B buszokra kerülnek. A 3-as ciklusban az ALU és az léptető dolgozik számára. Végül a 4-es ciklusban az eredményei eltárolódnak a regiszterekbe. Itt megjegyzendő, hogy a hardver négy része áll rendelkezésre, és minden egyes ciklus során egy adott utasítás csak az egyiket használja, ezzel felszabadítva a többi részt más utasítások számára.

A mi csővezetékes tervezetünket hasznos párhuzamba állítani egy gyári futószalaggal, amely autókat állít össze. Hogy elvonatkoztassuk a modell lényegét, képzeljük el, hogy minden percben elütnek egy nagy gongot, és ebben az időben minden autó egy állomással továbbmozdul a szalagon. A munkások minden egyes állomáson végrehajtanak valamilyen műveletet azon autón, amely éppen előttük áll, mint hozzáillesztik a kormánykereket vagy beszerelik a féket. Minden gongütéskor (1 ciklus) egy új autót tesznek a futószalag elejére, és egy kész autó gördül le a végéről. Eképpen, még ha többszáz ciklus, míg egy autó elkészül, akkor is, minden ciklusban egy teljes autó készen van. A gyár percenként egy autót elő tud állítani, függetlenül attól, hogy valójában meddig tart egy autó összeállítása. Ez a csővezeték ereje, és ez ugyanúgy vonatkozik CPU-kra, mint autógyárakra.

4.4.5 A hétszintes csővezeték: a Mic-4

Az egyik lényeges pont, amit elszépítettünk, az az a tény, hogy minden mikroutasítás kiválasztja a maga után következőt. A legtöbbjük csak a jelenlegi sorozatból válsztja ki a következőt, de az utolsó, mint csere6, gyakran egy többutas ágat képez, amely eldugítja a csővezetéket, mivel ez után lehetetlen folytatni az előre lehívást. Egy jobb módszert kell találnunk, hogy leküzdjük ezt a pontot.

A következő (és utolsó) mikroarchitektúra a Mic-4. A fő részeit a 4-35. ábra szemlélteti, de tekintélyes mennyiségű részletet kihagytunk az átláthatóság kedvéért. Mint a Mic-3, ez is rendelkezik IFU-val, amely előre lehív szavakat a memóriából és fenntartja a különféle MBR-eket.



4-35. ábra: A Mic-4 fő összetevői

Az IFU beérkező byte-áramlatot betölti egy új összetevőbe is, a **dekódoló egységbe (decoding unit)**. Ez az egység rendelkezik egy belső ROM-mal, amit az IJVM gépi utasítás indexel. Minden bejegyzés (sor) két részből áll: az IJVM utasításnak a hossza és egy index

egy másik ROM-ba, a mikro-művelet ROM-ba. Az IJVM utasítás hossza arra szolgál, hogy lehetővé tegye a dekódoló egység számára a beérkező byte-áramlat utasításokká tördelését, és így az mindig tudja, mely byte-ok a gépi utasítások és melyek az operandusok. Ha az aktuális utasításhossz 1 byte (pl. POP), akkor a dekódoló egység tudja, hogy a következő byte egy gépi utasítás. Ha azonban az aktuális utasításhossz 2 byte, akkor a

***[262-265]

Fordította: Pató Tímea
e-mail:h938745@stud.u-szeged.hu
262-265. oldal

a dekodoló egység felismeri, hogy a következő byte egy operandus, amit közvetlenül egy másik opcode követ. Mikor a WIDE előtag megjelenik, a következő byte átalakul egy speciális széles opcode-dá, pl: WIDE+ILOAD WIDE_ILOAD-dá válik.
A dekodóló egység az indexet elküldi egy mikroeljárásó ROM-ba, amit talált a táblázatban a következő komponensnek a soroló egységét (queueing unit-ot). Ez az egység néhány logikai plusz információt tartalmaz a két belső táblázathoz, egyet a ROM-ban, egyet a RAM-ban. A ROM tartalmazza a mikroprogramokat, amelyben minden egyes IJVM utasításnak van száma
az egymásra következő bejegyzésekről, amit mikroeljárásnak hívunk. A bejegyzések általában rendben vannak, így az a trükk, hogy pl: WIDE_ILOAD2 elágazás az ILOAD2-be, a Mic-2-ben nincs engedélyezve. Minden egyes IJVM sorozat esetenként kiíródik egy teljes duplasorozatba.
A mikroeljárások hasonlóak a Fig.4-5 mikroutasításaihoz, kivéve a NEXT_ADDRESS és JAM mezők hiányoznak és az új kódolt mezőknek szüksége van arra, hogy előírja az A busz bemenetét. A két új bit szintén adott: Final és Goto. A Final bit minden egyes IJVM mikro-eljárás sorozat utolsó mikroeljárásánál van beállítva, hogy jelezze azt. A Goto bit úgy van beállítva, hogy jelölje a mikroeljárásokat, ahol feltételes mikroelágazások vannak. Ezek formátumai különböznek a normál mikroeljárásoktól, pl: a JAM bitek és az index, amely a mikro-eljárású ROM-ba megy. A mikroutasításokat, amelyeket előzőleg hoztunk létre az adatmezővel és végrehajtottak egy feltételes mikroelágazást is, most be kell állítani két mikro-eljárásba.
A sorolóegység a következőképpen működik: kap egy mikroeljárású ROM indexet a dekódoló egységtől, aztán megkeresi a mikroeljárást és bemásolja egy belső sorba. Aztán bemásolja a következő mikroeljárást a sorba ugyanígy és így tovább. Ezt addig folytatja, amíg a Final bittel nem ütközik. Bemásolja ezt is, majd leáll. Feltéve, hogy ütközik egy mikroeljárás egy Goto bittel és még mindig van bőven hely balra a sorban, a dekódoló egység küld egy igazoló jelet vissza a dekódoló egységhez. Amikor a dekódoló egység érzékeli ezt a visszaigazolást, a következő IJVM utasítás indexét elküldi a soroló egységbe.
Ezen a módon az IJVM utasítás sorozat végül átváltozik mikroeljárások sorozatává a sorban. Ezek a mikroeljárások töltik a MIR-ekbe, amelyek jeleket küldenek, hogy kontrolálják az adatmezőt. Azonban van egy másik tényező, amelyet figyelembe kell venniük. A mezők az egyes mikroeljárásokban nem aktívak ugyanabban az időben. Az A és B mező aktív az első ciklus alatt, az ALU mező aktív a második ciklus alatt, a C mező a harmadik ciklus alatt aktív és a memória eljárásműveletek a negyedik ciklusban következnek be.
Ahhoz, hogy ezt amunkát megfelelően elvégezze, bevezetünk négy független MIR-t a Fig.4-35-ben. Minden egyes óraciklus kezdetén (Δw idő a Fig.4-3-ban) a MIR3 átmásolódik a MIR4-be. A MIR2 a MIR3-ba, a MIR1 pedig a MIR2-be és a MIR1-be

betöltődik egy új mikroeljárással a mikroeljárás sorból. Aztán minden egyes MIR előállítja a kontroll jeleit, de ezek közül csak néhányat használ. Az A és a B mezőt a MIR1-ből a regiszterek kiválasztására használják, hogy vezesse az A és B zárakat, de az ALU mezőt a MIR1-ben nem használják és nem kapcsolódik semmi máshoz az adatmezőben.
Egy óraciklussal később ez a mikroeljárás átmegy a MIR2-be és a regiszterek, amelyeket kiválasztott most biztonságosan az A és a B zárban vannak, hogy várják, hogy valami érdekes történjen. Ennek ALU mezőjét az ALU vezérlésére használjuk. A következő ciklusban a C mező visszaírja az eredményeket a regiszterekbe. Ezt követően átmegy a MIR4-be és elindítja a szükséges memóriaeljárásokat a most letöltött MAR (és MDR az íráshoz) felhasználásával.
Az utolsó ami a MIC-4-hez még szükséges: a mikroelágazások. Néhány IJVM utasítás, mint pl: IFLT, ahhoz szükséges, hogy feltételesen megalapozza az elágazást, amit N bitnek nevezünk. Ha egy mikroelágazás bekövetkezik a csőrendszer nem tud tovább működni. Ahhoz, hogy ezzel megbirkózzunk, egy Goto bitet adtunk a mikroeljáráshoz. Mikor a soroló egység ütközik a mikroeljárás során ezzel a bittel, amíg átmásolja a sorba, felismeri, hogy probléma van előtte és megakadályozza abban, hogy küldjön egy visszaigazolást a dekódoló egységnek. Ekkor a gép leáll ennél a pontnál, amíg a mikroelágazás feloldódik.
Valószínűleg néhány IJVM utasításon kívül már az eljárás is benne van a dekódoló egységben (de a soroló egységben nem), mivel nem küldte vissza az igazoló (azaz folytonos) jelet, amikor ütközött a mikroeljárás a Goto bittel. Speciális hardware és mechanizmusok szükségesek ahhoz, hogy letisztázzák ezt a zavart és visszatérjen az eredeti folyamathoz, de ez már nem ennek a könyvnek a témája. Mikor Edsger Dijkstra híres jegyzetét megírta a Goto állapot megfigyelt veszélyei címmel (Dijkstra, 1968), még nem tudta, hogy mennyire igaza volt.
Hosszú utat tettünk meg a Mic-1 óta. A Mic-1 egy egyszerű része volt a hardware-nek, majdnem a software-re jellemző. A Mic-4 egy nagymérvű csőrendszer tervezet 7 állapottal és jóval összetettebb hardware-rel. A csőrendszert sematikusan mutatja a Fig. 4-36, a bekarikázott számok feloldását, melynek elemeit a Fig. 4-35 tartalmazza. A Mic-4 automatikusan oda-visszaszállítja a byte sorozatot a memóriából, dekódolja azokat az IJVM utasításokhoz, konvertálja azokat a ROM használat mikroeljárás sorozatához és sorba rendezi a használathoz, ahogyan az szükséges. A csőrendszer első három állapota köthető az adatmező órájához, kívánság szerint, de ez nem mindig megfelelően működik. Például az IFU nem tud természetesen adagolni egy új IJVM opcode-t a dekódoló egységbe minden óraciklusban, mert az IJVM utasítások több ciklusra tagolódnak, alkalmazásra és kódra, melyek gyorsan telítődnek.

IFU→Decoder→Queue→Operands→Exec→Write back→Memory

Figure 4-36. A Mic-4-es csőrendszer

Minden egyes óraciklus alatt a MIR-ek előretolódnak és a mikroeljárások a sor alján bemásolódnak a MIR1-be, hogy elkezdjék az alkalmazást. Az ellenőrző jelek a négy MIR-ből aztán szétáramlanak az adatmezők között, amelyek a működés elindítását eredményezik. Minden egyes MIR ellenőrzi az adatmező egy különböző részét, ugyanígy különböző mikrolépéseit.
Ebben a tervezetben alaposan megvizsgáljuk a CPU csőrendszerét, amely engedélyezi az egyéni lépéseket, melyek nagyon rövidek, így az órafrekvencia magas. Sok CPU-t

terveznek alapvetően ilyen módon, különösen azokat, amelyek végrehajtják a régebbi utasítás készletet (CISC). Például a Pentium II-es végrehajtása koncepciójában hasonló a Mic-4 néhány szempontjában, mint ahogy ezt a következő fejezetben később látni fogjuk.

4.5 Fejlődő teljesítmény

A komputervállalatok arra törekednek, hogy rendszereik olyan gyorsan fussanak, amennyire csak lehetséges. Ebben a fejezetben számos kutatás alatt álló fejlett technikát ismerünk meg, melyek a rendszerteljesítmény (elsősorban CPU és memória) fejlesztését célozzák. A komputeriparban jelenlevő erős versenynek köszönhetően az időbeni eltérés olyan új kutatási ötletek, amelyek gyorsítják a számítógépeket és a megvalósulások között igen rövid. Következésképpen a szemléletek legtöbbje (amelyeket most fogunk megvitatni) már használatban vannak a létező termékek széles körében.

A szemléletek amiket megvitatunk, hozzávetőlegesen két kategóriára oszthatók: a teljesítmény fejlesztésekre és a gépi fejlesztésekre. A teljesítmény fejlesztések egy új CPU vagy memória építésének módszere, hogy a rendszert gyorsabbá tegyék anélkül, hogy változtatnának a gépen. A teljesítmény módosítása a gép változtatása nélkül azt jelenti, hogy a régi programok is futnak az új gépeken, ami egy fő eladási szempont. Egyik módja a teljesítmény növelésének, hogy gyorsabb órát alkalmaznak, de ez nem az egyetlen mód. A teljesítmény nyereség 80386-tól a 80486-on, a Pentium-on, a Pentium Pro-n, a Pentium II-n át a jobb kivitelezésnek tulajdonítható úgy , hogy a gépezet lényegében ugyanaz marad.
A fejlesztések néhány fajtáját csak a gépek változtatásával lehet megvalósítani. Néha ezek a változtatások előnyösek mint például utasítások vagy regiszterek hozzáadása, így a régi programok új modelleken is tudnak futni. Abban az esetben, ha egy teljesen új teljesítményt akarunk kapni, a software-t bizonyára változtatni kell, vagy legalább újra össze kell állítani egy új compiler-rel, hogy az új tulajdonságokból előnyt kovácsoljunk.
Azonban nagy ritkán a fejlesztők felismerték, hogy a régi gépezet túlélte használhatóságát és az egyetlen módja, hogy fejlesszék, ha az egészet újra kezdik. A RISC forradalom az 1980-as években egy ilyen áttörés volt, a másik a mai napig folyamatban van. Ezt egy példán keresztül is megismerjük (Intel IA-64) az 5. fejezetben.
Fejezetünk hátralevő részében, négy különböző technikát ismerünk meg a CPU teljesítmény növelésére. Három jól kiépített teljesítmény-fejlesztéssel kezdünk, aztán a negyediket is megismerjük, amihez egy kis technikai segítségre van szükség, hogy a legjobban működjön. Ezek a technikák: gyorsító tár (cache memory), elágazás előrejelzés (branch prediction), rendszeren kívüli alkalmazások regiszter újra nevezéssel (out-of-order execution with register renaming) és az elméleti alkalmazás (speculative execution).

4.5.1 Gyorsító tár (cache memory)

A történelem folyamán a komputer tervezés egyik legnagyobb kihívása, hogy képesek legyenek létrehozni egy olyan memóriarendszert, ahol a processzorok operandus ellátása a lehető leggyorsabb. A processzsebesség jelenlegi magas szintje nem jár

együtt megfelelő sebesség növekedéssel a memóriákban. A CPU-hoz képest a memóriák egyre lassulnak évtizedek óta. Mivel az elsődleges memóriának nagy jelentősége van, ez nagy mértékben korlátozza a magas teljesítményű rendszerek fejlődését és arra ösztönzi a kutatást, hogy számos módon sort kerítsenek a memória sebesség probléma megoldására, mivel ez sokkal lassabb a CPU sebességénél, vagyis kb. egyre rosszabb minden évben.

A modern processzorok nagy igénybe vételt jelentenek a memória rendszer számára, úgy mint a késlekedés időszakai (késlekedés az operandus betöltése során) és a sávszélesség (az adatmennyiség időegység alatti betöltése). Sajnos ezzel a két tényezővel a memória rendszerben igen hadilábon állunk. Sok technika van sávszélesség növelésére, de ez csak a késlekedés növelésével érhető el. Például azok a csőrendszer technikák amelyeket a Mic-3-ban használnak, olyan memóriarendszert alkalmaznak, amely memóriája multiprogramozható, átlapozható és hatékonyan kezeli a kéréseket. Sajnos, mint a Mic-3-nál, ennek következménye, hogy nagyobb késedelmet okoz az egyéni memóriaeljárások során. Ahogy a processzor idősebessége növekszik, egyre bonyolultabbá válik, hogy a memóra rendszert ellássa a betöltött operandusokkal egy vagy két óraciklus alatt.

A probléma megoldás egyik módja a tárak megosztása. Mint ahogy a 2.2.5. fejezetben láttuk, a tár egy kicsi, gyors memóriában tartja a legutóbb használt memóriaszavakat, növelve ezzel az elérési sebességet. Ha a szükséges memóriaszavak igen nagy százalékban vannak a tárban, akkor a tényleges memória késlekedése nagymértékben csökken.

A sávszélesség és a késedelem fejlesztésének egyik hatékony technikája, hogy multiprogramozható tárakat használunk. Az alaptechnika, hogy a gyakran használatos szavakat egy elkülönített tárban tároljuk az utasítások és adatok számára. Több előnye is van, hogy elkülönítjük a tárakat az utasítások és adatok számára, amit gyakran elkülönített tárnak (split cache-nek) hívunk. Először is a memória-eljárások függetlenül elindulhatnek minden egyes tárban, a memóriarendszer sávszélessége ténylegesen megduplázódik. Ez az amiért ésszerű 2 különálló memóriaportot teremteni, ahogy ezt a Mic-1-ben is láttuk, hogy minden egyes portnak külön tára legyen. Megjegyezzük, hogy minden egyes tár külön képes elérni a főmemóriát.

Manapság sok memóriarendszer sokkal bonyolultabb, mint ez, és egy pótlólagos tár, amit 2.szintű tárnak (level 2 cache-nek) hívunk, amely az utasítás- és adattárak valamint a főmemória között van. Valójában három vagy többszintű tárak is lehetnek, attól függően, ahogy az igényesebb memóriarendszerek megkövetelik. A Fig. 4-37-ben egy háromszintű tárral ellátott rendszert láttunk. A CPU chip maga is tartalmaz kis utasítás tárat és egy kis adattárat, általában 16-tól 64 KB-ig. Van még kétszintű tár, amely nincs a CPU chipben, de tartalmazhatja ezt a CPU csomag, közvetlenül a CPU chip mellett és kapcsolódva ehhez egy nagy sebességű mezőn keresztül. Ez a tár általában..

***[266-269]

Az L2 cache tipikus mérete 512 kB és 1 MB közötti. A 3. szintű cache a processzoron található és egy néhány MB SRAM-ot tartalmaz, amely sokkal gyorsabb, mint a fő DRAM memória. A cache-ek többnyire összetettek, az 1-es szintű cache teljes tartalma a 2-es szintű cache-ben van, a 2-es szintű cache-t viszont a 3-as tartalmazza.
A cache-ek két fajta címzési módtól függenek ahhoz, hogy elérjék a céljukat. **Térbeli elhelyezkedése** azon a megfigyelésen alapul, hogy a memóriahelyek, amelyek számszerűleg azonos címmel rendelkeznek, mint egy már lefoglalt memóriahely, szívesen lesznek lefoglalva a közeljövőben. A cache-ek ezt a tulajdonságot az által érik el, hogy több adatot használnak fel, mint amire kérték, azzal a reménnyel, hogy a jövőbeni adatkérelmeket előre láthatják. Az **ideiglenes elhelyezés** akkor fordul elő, ha a már lefoglalt memóriahelyeket ismét lefoglalják. Ezt okozhatják például memóriahelyek közel a verem tetejéhez vagy utasítások egy ciklus belsejében. Az ideiglenes helyeket elsősorban azért tervezték, hogy biztosítsák egy cache-tévesztés elutasításának lehetőségét. Számos cache-helyettesítő algoritmus ideiglenes helyeket hoz létre azáltal, hogy visszautasítja azokat a bejegyzéseket, melyeket előzőleg nem értek el.
Az összes cache a következő modellt használja. A fő memória fix blokkokra van osztva, melyeket cache-vonalaknak neveznek. Egy cache-vonal tipikusan 4-64 egymás után álló byte-ból áll. A vonalak számozása 0-nál kezdődik egy 32 byte-os vonalmérettel, a 0. számú vonal 0.-31. byte-ig tart, az 1-es számú 32.-63.-ig stb. Bármelyik időpillanatban néhány vonal a cache-ben van. Amikor a memóriára hivatkozunk, a cache-t irányító áramkör leellenőrzi, hogy a hivatkozott szó jelenleg a cache-ben van-e. Ha igen, ez az érték használható, megspórolva ezzel egy utat a fő memóriához. Ha a szó nincs ott, néhány vonalbejegyzés törlésre kerül a cache-ből, és a szükséges vonal betöltésre kerül a memóriából, vagy az alacsonyabb szintű cache-ből. Számos variáció létezik erre, de mind közül az ideális az, hogy amennyire csak lehet, meg kell tartani a leggyakrabban használt vonalakat a cache-ben ahhoz, hogy maximalizáljuk a teljesített memóriahivatkozások számát a cache-ből.

Közvetlen leképzésű (Direct-Mapped) **cache-ek**

A legegyszerűbb cache közvetlen leképzésű cache-ként ismeretes. Példaként egy egyszeres szintű közvetlen leképzésű cache látható a 4-38. ábrán. Ez a példa-cache 2048 bejegyzést tartalmaz. Minden bejegyzés (sor) pontosan egy cache-vonalat tartalmazhat a fő memóriából. Egy 32 byte-os cache-vonalmérettel (ezen a példán) a cache 64 kB-ot tartalmazhat. Mindegyik cache-bejegyzés három részből áll:

1. A Valid bit jelzi, hogy ebben a bejegyzésben van-e érvényes adat. Amikor a rendszer indul, az összes bejegyzés invalid értéket kap.
2. A Tag mező egy egyedülálló, 16 bites értéket tartalmaz, mely azonosítja a megfelelő memóriavonalat, amelyből az adat származik.
3. A Data mező a memóriában található adat másolatát tartalmazza. Ez a mező egy 32 byte-os cache-vonalat tartalmaz.

Egy közvetlen elérésű cache-ben egy adott memóriaszó pontosan egy helyen tárolható a cache-ben. Adott egy memóriacím és csak egy hely van, ahol a cache-ben kereshetjük. Ha nincs ott, akkor nincs a cache-ben. A cache-ben történő tárolás és adatkinyerés érdekében a címet négy részre törik, amit a 4-38.(b) ábra mutat.

1. A TAG mező egy cache-bejegyzésben tárolt tag bitekhez kapcsolódik.
2. A LINE mező jelzi, mely cache-bejegyzés tartalmazza a kapcsolódó adatot, ha egyáltalán jelen van.
3. A WORD mező elárulja, mely szóra hivatkozunk a vonalon belül.
4. A BYTE mezőt általában nem használják, de ez csupán egy byte-ból áll. Ez azt jelzi, hogy a szón belüli melyik byte szükséges. A csak 32 bites szavakat támogató cache számára ez a mező mindig 0 lesz.

Amikor a CPU feldolgoz egy memóriacímet, a hardware kicsomagolja a vonal 11 bitjét a címből és felhasználja ezeket a cache-ben található 2048 bejegyzés egyikének megtalálására. Ha a bejegyzés érvényes, a memóriacím Tag mezőjét és a cache-bejegyzés Tag mezőjét összehasonlítja. Ha egyeznek, a cache-bejegyzés által tárolt szó lesz felhasználva. Ezt nevezik cache-találatnak (cache-hit). Egy találaton a beolvasott szót a cache-ből veszi, kiküszöbölve a memóriához fordulás szükségét. Csak az éppen szükséges szó kerül kinyerésre a cache-bejegyzésből. A bejegyzés megmaradt része nem használatos. Ha a cache-bejegyzés érvénytelen vagy a tagok nem egyeznek, ekkor a szükséges bejegyzés nincs jelen a cache-ben. Ezt nevezik cache-tévesztésnek (cache-miss). Ebben az esetben a 32 byte-os cache-vonal lehívódik a memóriából és a cache-entry-ben tárolódik, felülírva annak előző tartalmát. Azonban ha a létező cache-bejegyzés a betöltés óta módosításra kerül, ennek tartalmát vissza kell írni a fő memóriába, mielőtt törölnénk.

A döntés összetettségének ellenére a szükséges szó elérése figyelemreméltóan gyors. Amint a cím ismeretes, a szó pontos helye ismertté válik, ha ez jelen van a cache-ben. Ez azt jelenti, hogy lehetséges egy szót kiolvasni a cache-ből és a processzorhoz szállítani ugyanazon idő alatt, amíg meghatározásra kerül a szó helyessége (tagok összehasonlításával). Így a processzor pontosan egy szót kap a cache-ből egyidejűleg, vagy lehetőség szerint mielőtt (még) a szóról bebizonyosodik, hogy ez a keresett szó.

A mappelési séma (sablon) egymást követő memóriavonalakat helyez az egymást követő cache-bejegyzésekbe. Ekkor maximum 64 kB folyamatos adatot tárolhatunk a cache-ben. Azonban két vonal, mely eltér a 64 kB-os precíz címükben vagy ennek a számnak a többszörösében, nem tárolható a cache-ben ugyanabban az időben (mert azonos a Line-értékük). Például, ha egy program egy helyen lévő adatot elér, majd futtat egy utasítást, melynek szüksége van az x+64,536 helyen lévő adatra (vagy bármely ugyanazon vonalon belüli helyen), a második utasítás a cache-bejegyzés újbóli betöltésére késztet, felülírva annak előző tartalmát. Ha ez elég gyakran megtörténik, gyenge teljesítményt okoz. Ebben az esetben egy cache rossz magatartása rosszabb, mintha egyáltalán nem lenne cache, és minden egyes memóriaműveletkor egy szó helyett egy teljes cache-vonalat olvasnánk be.

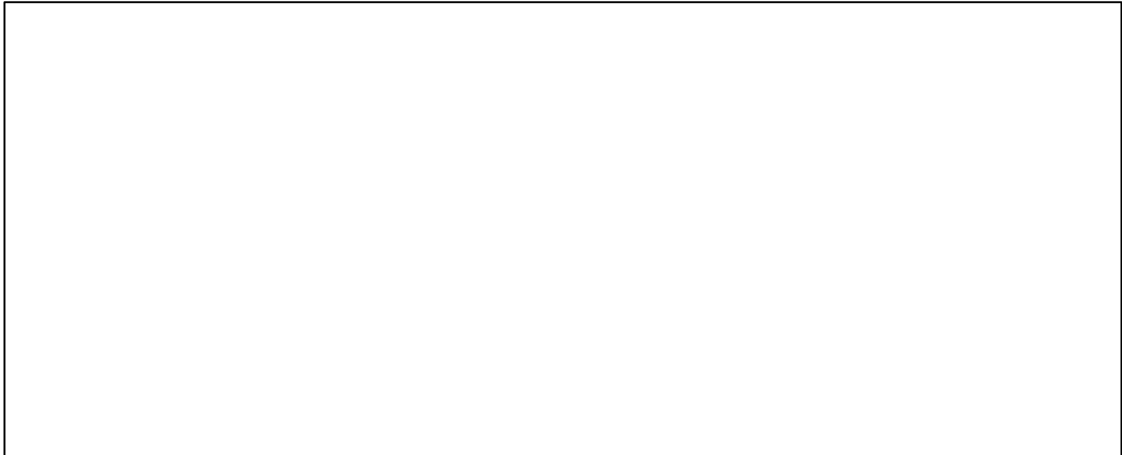
A direkt mappelésű cache-ek a legáltalánosabban elterjedt cache-fajták, elég hatásosan működnek, mert a fent leírt ütközések meglehetősen ritkán, vagy egyáltalán nem fordulnak elő. Például egy nagyon okos fordító csupán számottevő ütközéseket produkál, amikor utasításokat és adatokat helyez el a memóriában. Megjegyezzük, hogy a részletesen leírt eset nem fordulna elő egy olyan rendszerben, amely külön utasítás- és adat-cache-sel rendelkezik. Azért, mert a kéréseket különböző cache-ek szolgálják ki. A két cache-nek tapasztalhatunk egy másik előnyét: rugalmasabb konfliktuskezelés a memóriamintákkal.

Csoport-asszociatív cache:

Mint fent említettük, a memóriában sok különböző vonal versenyez ugyanazokért a

cache-cellákért. Ha egy program a 4-38.(a) ábrán lévő cache-t használja, akkor a 0 és a 65536 címen lévő szót használja, ekkor állandó konfliktusa lesz minden egyes bejegyzéssel, melyek potenciálisan kilakoltatják a másikat a cache-ből. A megoldás erre a problémára az, hogy lehetővé kell tenni kettő vagy több vonalat minden egyes cache-bejegyzésben. Egy cache n lehetséges bejegyzéssel minden egyes címen az n utas csoport asszociatív cache-nek nevezzük. Egy 4-utas csoport asszociatív cache-t ábrázol a 4-39. ábra.

Egy csoport asszociatív cache következésképpen összetettebb, mint egy közvetlen leképzésű cache, mert bár a helyes cache-bejegyzés a hivatkozott memóriacímből kerül kiszámításra, az n db cache-bejegyzés-halmazt kell leellenőrizni ahhoz, hogy a szükséges vonal jelen van-e. Azonban a gyakorlat azt mutatja, hogy a kétutas és a négyutas cache-ek elég jók ahhoz, hogy megtegyék ezt az extra áramköri lépést. Egy csoport asszociatív cache használata egy választást jelent a tervező számára. Mikor egy új bejegyzés kerül a cache-be, mely jelen-tartalmat töröljük? Az optimális döntés természetesen a jövőben születik, de egy elég jó algoritmus minden esetre az LRU (legkevésbé mostanában használt).



4-37. ábra. Egy rendszer 3 szintű cache-sel

Bejegyzés	Valid	Tag	Data	Címek, amelyek használják ezt a bejegyzést
2047				65504-65535, 131040-131072
…				
4				
3				
2				64-95, 65600-65631, 131036-131067, …
1				32-63, 65568-65599, 131004-131035, …
0				0-31, 65536-65567, 130972-131003, …

4-38.(a) ábra. Egy közvetlen elérésű cache

16 bit	11 bit	3 bit	2 bit
TAG	LINE	WORD	BYTE

4-38.(b) ábra. Egy 32 bites virtuális cím

	Valid	Tag	Data	Valid	Tag	Data	Valid	Tag	Data	Valid	Tag	Data
2047												
…												
4												
3												
2												
1												
0												
		Entry A			Entry B			Entry C			Entry D	

4-39. ábra. Egy négyutas asszociatív cache

***[270-273]

270-273. oldal, 4. fejezet - A microarchitektúra szint, 4.5. rész - A teljesítmény növelése

Ez az algoritmus tartja a sorrendjét minden olyan rekeszhalmaznak, amely hozzáférhető a megadott memóriarekeszből. Amikor a jelenlévő vonalak bármelyikéhez hozzáférés történik, az algoritmus frissíti a listát, minek során az adott bejegyzést jelöli meg legutoljára hozzáfértként. Ha eljön az ideje egy bejegyzés lecserélésének, akkor a lista végén lévő - vagyis a legrégebben használt - bejegyzést selejtezi le.

A végsőkig folytatva, lehetséges egy olyan 2048 utas gyorsítótár /cache/ is, amely egy egyedüli, 2048 vonalbemenetet tartalmazó halmazból áll. Ilyenkor az összes memóriacím erre a halmazra képez le, tehát a keresőnek össze kell hasonlítania a címet mind a 2048, cacheben lévő címkével. Megjegyzendő, hogy jelen esetben minden bemenetnek kell hogy legyen címke-összehasonlító logikája. Mivel a VONAL mező 0 hosszúságú, a CÍMKE mező lesz a teljes cím, a SZÓ és a BYTE mezőket kivéve. Ráadásul, amikor egy cache vonal lecserélődik, mind a 2048 rekesz fellép lehetséges helyettesítőként. Egy 2048 bejegyzésből álló, rendezett lista fenntartásához nagyon sok könyvelés szükséges, ami az LRU /legrégebben használt/ helyettesítést kivitelezhetetlenné teszi. (Ne feledjük azt, hogy ezt a listát minden memória műveletkor aktualizálni kell, nem csak hiba esetén). Meglepő, de legtöbbször a magas szintűen társítható gyorsítótárak sem növelik sokkal jobban a teljesítményt, mint az alacsony szintűek, sőt, néhány esetben rosszabbul is működnek. Ezen okokból, a 4 utasnál nagyobb társíthatóság egészen ritka.

Végül, az írások a gyorsítótárak egy speciális problémáját is felvetik. Amikor egy processzor egy szót ír, és a szó a cacheben van, akkor nyilvánvalóan vagy frissítenie kell a szót, vagy érvénytelenítenie kell a cache bejegyzést. Csaknem az összes konstrukció cache-t frissíti. De mi a helyzet a fő memóriában lévő másolattal? Ez a művelet elhalasztható addig, amíg a cache vonal készen nem áll az LRU algoritmus által vezérelt lecserélésre. Ez a megoldás nehézkes, de egyik lehetőség sem részesíthető igazán előnyben. A fő memória azonnali frissítését közvetlen írásnak /write through/ nevezik. Ez a megközelítés többnyire egyszerűbben megvalósítható, és megbízhatóbb, mivel a memória mindig friss adatokat tartalmaz - és jól jön, ha például hiba történik, és szükségessé válik a memória állapotának visszaállítása. Sajnos, ez általában nagyobb memória írási forgalmat igényel, ezért a kifinomultabb megvalósításokban olyan alternatív megoldások alkalmazására törekszenek, mint a késleltetett írás /write deferred/, vagy a vissza írás /write back/.

Egy ezekkel kapcsolatos probléma fűzhető még az írásokhoz : mi történik, ha az írás olyan helyre történik, ami éppen nincs cache-elve? Be kellene vinni az adatot a cache-be, vagy csak ki kellene írni a memóriába? Ez ismét olyan helyzet, amikor egyik választás sem lehet mindig a legjobb. A legtöbb olyan konstrukció, amely késlelteti a memóriába írást azon van, hogy írás kihagyás alatt vigye be az adatot a cache-be, ezt a technikát hívják írás kiosztásnak /write allocation/. Másrészt viszont a közvetlen írást alkalmazóak pont arra törekszenek, hogy ne kelljen kiosztani egy bejegyzést írásnak, mert ez a megoldás bonyolítaná az amúgy egyszerű konstrukciót. Az írás kiosztás tehát csak akkor győzhet, ha léteznek ismétlődő írások ugyanolyan,

vagy éppen különböző szavakra a gyorsítótár vonalában.

4.5.2 Elágazás előrejelzés

A modern számítógépek nagyon sok adatcsatornát/csővezetéket tartalmaznak. A 4-35. ábrán látható adatcsatorna hét szakaszból áll; a csúcstechnikájú számítógépeknek néha 10, vagy még több szakaszból álló csatornarendszerük van. A csővezetékek alkalmazása lineáris kódolással működik a legjobban, ilyenkor a lehívási egység egyszerűen csak kiolvassa az egymást követő szavakat a memóriából, és kiküldi azokat a dekódoló egységnek, még mielőtt szükség lenne rájuk.
Az egyetlen kisebb probléma ezzel a nagyszerű modellel az, hogy a köze sincs a valósághoz. A programok nem lineáris kódsorozatok. Teli vannak elágazási utasításokkal. Gondoljuk át a 4-40(a) ábra egyszerű utasításait, amelyek az i változó 0-hoz hasonlítását oldják meg (a gyakorlatban talán ez a legáltalánosabb teszt). Az eredménytől függően, egy másik változó, a k felveszi a két lehetséges érték egyikét.

```
if (i==0)          Then:   CMP i,0      ;i összehasonlítása 0-val
    k=1;           Else:   BNE Else     ;elágazás Else-hez, ha nem egyeznek
else               Next:   MOV k,1      ;1 megy k-ba
    k=2;                   BR Next      ;Feltétel nélküli elágazás Next -be
                           MOV k,2      ;2 megy k-ba
    (a)                       (b)
```

4-40. ábra. (a) Egy programrészlet. (b) A programrészlet általános assembly nyelvre lefordítva.

Egy lehetséges assembly nyelvre való fordítás látható a 4-40(b) ábrán. Az assembly nyelvet a későbbiekben fogjuk tárgyalni ebben a könyvben. és most nem is fontosak a részletek, de a géptől vagy a fordítótól függően, egy többé-kevésbé olyan kód, mint ami a 4-40(b) ábrán van, megfelelő. Az első utasítás hasonlítja össze i-t 0-val. A második elágazik az Else címkéhez, ami a különben utasítássorozat kezdete, ha i nem 0. A harmadik utasítás 1-et rendel k-hoz. A negyedik egy elágazás a következő utasításhoz. A fordító arra alkalmas módon elhelyezett egy Next nevű címkét, így tehát van hová elágazni. Az ötödik sor 2-t rendel k-hoz.

Ami itt megfigyelhető, az az, hogy az öt utasításból kettő elágazás. Továbbá, ezek közül az egyik - BNE - egy feltételes elágazás (egy olyan elágazás, amely csak akkor történik meg, ha a valamilyen feltétel teljesül, jelen esetben ha az előző CMP utasítás két operandusa megegyezik). Tehát itt a leghosszabb lineáris kódsorozat két utasításból áll. Mindezekből következik, hogy az utasítások nagy sebességű lehívása és betöltése az adatcsatornákba nehezen megoldható dolog.

Első pillantásra úgy tűnhet, hogy az olyan feltétel nélküli elágazások, mint a BR Next utasítás a 4-40(b) ábrán, nem jelentenek problémát, hiszen nincs semmi kétértelműség abban, hogy hová kell menni. Miért ne tudná a lehívási egység egyszerűen tovább olvasni az utasításokat a célcímtől (az a hely, ahová az elágazás történik) kezdve?

A baj a csővezetékek alkalmazásának természetéből fakad. A 4-35. ábrán például láthatjuk, hogy az utasítás dekódolása a második szakaszban történik. Így a lehívási

egységnek el kell döntenie, hogy honnan kell a következő lehívást végrehajtania, még mielőtt tudná, hogy épp miféle utasítást kapott. Csak egy ciklussal később tudhatja meg, hogy éppen egy feltétel nélküli elágazást vett fel, de ekkorra már elkezdte a feltétel nélküli elágazást követő utasítás lehívását. Következésképpen tekintélyes számú, csővezetékeket alkalmazó gépnek (mint például az UltraSPRC II) megvan az a tulajdonsága, hogy végrehajtja a feltétel nélküli utasítás utáni utasítást, még akkor is, ha logikailag nem kellene. Az elágazás utáni helyet késleltetési résnek /delay slot/ nevezik. A Pentium II (és a 4-40(b) ábrán használt gép) nem ilyen, de a probléma megkerülésének belső összetettsége gyakran rendkívül bonyolult. Egy optimizáló fordítóprogram meg fog próbálni valami hasznos utasítást elhelyezni a késleltetési résbe, de gyakran nem áll semmi a rendelkezésére, így kénytelen lesz egy NOP utasítást beszúrni oda. Ha így tesz, a program ugyan helyes marad, de nagyobbá és lassabbá válik.

A feltétel nélküli elágazások bosszantóak, de a feltételesek még rosszabbak. Nem elég, hogy ezeknek is van késleltetési résük, de ilyenkor a lehívási egység hosszú ideig nem tudja, honnan kell olvasnia az adatcsatornából. Eleinte a csővezetékeket alkalmazó gépek egyszerűen leálltak addig, amíg megtudták, hogy mi a teendő az elágazásnál. A három-négy cikluson át tartó megakadás pedig, különösen ha az utasítások 20%-a feltételes elágazás, rombolja a teljesítményt.

Ennélfogva a legtöbb gép, amikor feltételes utasítással találkozik, megpróbálja előre jelezni, hogy az elágazást követni kell-e, vagy sem. Jó lenne, ha egyszerűen csak behelyezhetnénk egy kristálygömböt valamelyik üres PCI aljzatba, hogy segítsen a jóslásban, de mindezidáig ezt a megoldást nem koronázta siker.

Egy ilyen periféria hiányában, az előrejelzés elvégzésének változatos módjait agyalták ki. Az egyik nagyon egyszerű módszer a következő : feltételezzük, hogy az összes visszafelé mutató feltételes elágazást követni kell, az előre mutatókat pedig nem. Az első részt az indokolja, hogy a visszafelé mutató elágazások gyakran egy ciklus végén helyezkednek el. A legtöbb ciklus többször hajtódik végre, tehát általában jó ötlet arra fogadni, hogy egy ciklus elejére mutató elágazást visszaugrás követ.

A második rész egy picit rázósabb. Az előre mutató elágazások némelyike akkor követendő, ha bizonyos hibafeltételek teljesülnek a software-ben (pl. egy file-t nem lehet megnyitni). A hibák ritkák, tehát a legtöbb velük társítható elágazást nem kell követni. Természetesen, sok olyan előre mutató elágazás van, aminek nincs köze a hibakezeléshez, tehát a siker aránya nem olyan magas, mint a visszafelé mutató elágazásoknál. De még ha nem is fantasztikusan jó, azért ez a szabály is jobb, mint a semmi.

Ha egy elágazást helyesen előre jelzett, nincs semmi különös tennivaló. A végrehajtás a célcímtől folytatódik. A baj akkor jelentkezik, ha hibás az előrejelzés. A neheze azon a már végrehajtott utasítások érvénytelenítésében rejlik, amelyeket mégsem kellett volna elvégezni.

Ennek kétféleképpen lehet nekilátni. Az első módszer lényege, hogy engedni kell az előre jelzett feltételes elágazásokat követő lehívott utasítások végrehajtását

mindaddig, amíg nem próbálnak változtatni a gép állapotán (pl. tárolás egy regiszterbe). A regiszter felülírása helyett a kiszámított érték egy (titkos) munkaregiszterbe kerül, és csak azután másolódik az igazi regiszterbe, miután kiderült, hogy az előrejelzés helyes volt. A második módszer bármely felülírásra kerülő regiszter előző értékének feljegyzésén (pl. egy titkos munka regiszterben) alapul. Így a gép visszaállítható a hibás előrejelzés idején fennálló állapotába. Mindkét megoldás összetett és ipari mennyiségű könyvelést igényel a helyes működéshez. És ha egy második feltételes elágazás is beüt, még mielőtt ismerté válna, hogy az elsőre vonatkozó előrejelzés helyes volt-e, a dolgok nagyon zavarossá válhatnak.

Dinamikus elágazás-előrejelzés

Nyilvánvaló, hogy a pontos előrejelzés nagy érték, hiszen lehetővé teszi, hogy a CPU teljes sebességgel haladjon. Ebből következik, hogy jelenleg nagyon sok kutatás folyik az elágazás előrejelzés fejlesztésén. (pl. Driesen és Holzle, 1998; Juat et al., 1998; Pan et al., 1992; Sechrest et al., 1996; Sprangle et al., 1997; és Yeh és Patt, 1991). Az egyik megközelítés az, hogy a CPU fenntart egy előzmény táblázatot (egy speciális hardwareben), amelyben naplózza a megtörtént feltételes utasításokat, így ha újra fellépnek, utánuk lehet nézni. Ennek a rendszernek a legegyszerűbb változata a 4-41(a) ábrán látható. Itt az előzménytábla tartalmaz egy bejegyzést az összes feltételes elágazás utasításról. A bejegyzésben szerepel az elágazás utasítás címe, és egy olyan bit, amely megmutatja, hogy legutóbbi végrehajtáskor követni kellet-e. Ezt a rendszert használva az előrejelzés egyszerűen annyiból áll, hogy az elágazás ugyanúgy fog működni, mint a megelőző fellépésekor. Ha az előrejelzés helytelen, az előzmény táblázatban lévő bit megváltozik.

/az ábrán előforduló kifejezések : rés, érvényes, elágazás cím/címke, elágazás/nem elágazás, előrejelző bitek, célcím/

4-41. Ábra. (a) Egy 1 bites elágazás előzménytábla. (b) : Egy 2 bites elágazás előzménytábla. (c) : Elágazás utasítás címek és célcímek közötti leképezés.

Több más módja is van az előzménytábla megszervezésének. Valójában ezek ugyanazok a módszerek, mint amit a cache szervezéshez szoktak használni. Vegyünk egy gépet, olyan 32 bites utasításokkal, amelyek szóhatárra igazítottak, vagyis minden memóriacím 2 alsó bitje 00. Egy közvetlenül leképezett előzménytáblában, ami 2n bejegyzést tartalmaz, az elágazás utasítás alsó n+2 bitje eltávolítható, és eltolható jobbra két bittel. Ez az n-bites szám indexként használható az előzménytáblában ott, ahol az eltárolt cím és az elágazás címének egyezés ellenőrzése történik. Csakúgy, mint a cache esetében, itt sincs szükség az alsó n+2 bit tárolására, tehát kihagyhatóak (pl. csak a felső címbitek - a címke - kerülnek eltárolásra). Ha egyezés van, az előre jelző bit használható az elágazás előrejelzésére. Ha a rossz címke van jelen, vagy hibás a bejjegyzés, hiány lép fel, pont, mint a cache esetében. Ebben az esetben az előre/hátra mutató elágazások szabálya alkalmazható. Ha az elágazás táblázatnak mondjuk 4096 bejegyzése van, akkor a 0,16384,32768,... című elágazásoknál ellentmondás lesz, hasonlóan a cache ugyanezen problémájához.

Szikora Roland
h837145@sirius.cab.u-szeged.hu

***[274-277]

...ugyanez a megoldás lehetséges: kétirányű, négyirányú, vagy n-irányú társhivatkozás, ahol a felugyelet teljes társíthatóságára van szükség.

Elég nagy tábla és széleskörű társíthatóság esetén ez a módszer a legtöbb esetben jól működik. Azonban egy probléma mégis adódhat. Amikor végleg kilépünk egy hurokból, az utolsó elágazás rosszul lesz megállapítva, és ami mégrosszabb, ez meg fogja változtati a bitet a history táblában, ami miatt az legközelebb "nincs elágazás"-t fog mutatni. Amikor egy új hurok kerül végrehajtásra, az első iteráció végén levő elágazás rosszul lesz előrejelezve. Ha ez a hurok egy másik hurokban, vagy egy gyakran meghívott eljárásban található, ez a hiba gyakran előfordul.

Hogy kiküszöböljük ezt a problémát, megadhatjuk a táblabejegyzést egy második lehetőségnek. Ha ezt a módszert használjuk, az előrejelzés csak két egymást követő hibás előrejelzés után fog megváltozni. Ebben a megközelítésben két bitre van szügségünk a historyban: az egyik azt mutatja, hogy az elágazásnak mit "kellene" csinálnia, a másik pedig azt, hogy mit csinált utoljára. (4-41(b) ábra)

Egy másik módszer erre az algoritmusra az, hogy vizsgáljuk egy véges állapottal rendelkező gép négy állapotát, mint ahogy az a 4-42. ábrán is látható. Az egymást követő helyes "nincs elágazás" előrejelzések után az FSM a 00 állapotba kerül és "nincs elágazás"-t fog mutatni a következő alkalommal. Ha ez az előrejelzés hibás, átvált 10-ra, de még mindig azt fogja mutatni, hogy nincs elágazás. Ha ez az előrejelzés is hibás, 11-es állapotba kerül, és legközelebb már "elágazás"-t fog mutatni. Tehát a baloldali bit az előrejelzés, a jobboldali pedig azt mutatja, hogy mi történt valójában. Bár ez a példa csak 2 bittel dolgozik, lehetséges olyan felépítés, ami 4 vagy 8 bitet használ.

No branch: nincs elágazás
Predict: no branch: előrejelzés: nincs elágazás
Predict: branch: előrejelzés: elágazás
Branch: elágazás
Predict: no branch one more time: előrejelzés: ismét nincs elágazás
Predict: branch one more time: előrejelzés: újabb elágazás

4-42. ábra: Egy 2-bites véges állapotú gép elágazáselőrejelzésre

Ez nem az első FSM-ünk, hiszen a 4-28. ábrán már láthattunk egyet. Tulajdonképpen mindegyik mikroprogramunk felfogható egy-egy FSM-nek, ahol minden sor egy -a gép által felvehető- állapotot jelöl. Az FSM-ek nagyon széles körben használhatóak a hardwaretervezés minden területén.

Korábban feltételeztük azt, hogy minden feltételes elágazás célja ismert, mint például egy egyértelműen megadott cím egy elágazásra (ami magát az utasítást is tartalmazza) vagy egy relatív értékre az aktuális utasításból (pl: egy előjeles számot hozzá kell adni a programszámlálóhoz.). Ez a feltételezés a legtöbbször igaz, de néhány feltételes utasítás a regiszterekkel végzett matematikai művelet eredményére ugrik. Még ha a 4-42. ábra FSM-je pontosan megállapítja, hogy az elágazást végre kell hajtani, tem tudja megtenni, mert a cél ismeretlen. Az egyik lehetséges megoldás az, hogy az utolsó elágazás célcímét is eltároljuk a history táblába, mint ahogy az a 4-41c. ábrán látható. Így ha a táblában az található, hogy az 516-os címen levő elágazás a 4000-es címre ugrott, akkor ha elágazás következik, az előrejelzés "elágazás 4000-re" lesz ismét.

Egy másik lehetőség az, hogy eltároljuk az utolsó *k* elágazást függetlenül attól, hogy mik voltak az utasítások (Pan et al., 1992). Ez egy *k*-bites szám, amit az **elágazás előzményváltó regiszterben** (branch history shift register). Ezt a *k*-bites kulcsot párhuzamosan osszehasonlítjuk az előzménytábla összes bejegyzésével, és ha egyezést találunk, az itt talált előrejelzést fogjuk használni. Ez a technika elég jól működik a gyakorlatban.

Statikus elágazáselőrejelzés

A korábban ismertetett elágazáselőrejelző technikák azaz a program futása közben változtak. Az előnyük az, hogy így képesek voltak alkalmazkodni a program aktuális állapotához. Viszont van egy hátrányuk is, megpedig az, hogy bonyolult és drága hardware-re van szügség a használatukhoz.

Lehetőség van arra, hogy a fordítóprogram is segítsen. Pédául akkor, amikor a compiller egy ilyen kifejezéssel találkozik:

```
for (i=0; i<10000000; i++) {...}
```

amelyről tudjuk, hogy a hurok végén található ugrás nagyon gyakran hajtódik végre. Ha ezt meg tudjuk értetni a géppel, rengeteg erőfeszítést tudunk megspórolni.

Bár ez egy felépítésbeli változás, néhány gép, mint amilyen például az UltraSPARC II is, rendelkezik a feltételes elágazások utasításainak egy másik készletével is a hagyományosok mellett (amikre a visszafele kompatibilitás miatt van szükség). Az új gépeknél van egy bit, amit a fordító tud változtatni attól függően, hogy lesz-e elágazás, vagy sem. Amikor egy ilyen bitbe ütközünk, a feldolgozó egységnek csak azt kell tennie, amit az mond. Továbbá így nincs szükség arra, hogy az előzménytáblában értékes helyet vesztegessünk el ezekre az utasításokra, ezáltal csökkenteni tudjuk az esetlegesen előforduló hibákat.

Végül az utolsó elágazáselőrejelző módszerünk a körvonalazáson alapul (Fisher és Freudenberger, 1992). Ez is egy statikus elágazáselőrejelző technika, de itt inkább a compiller próbálja kitalálni, hogy melyik elágazást kell végrehajtani és melyiket nem, miközben a program fut (például egy szimulátoron), figyelve az elágazások viselkedését..Ezt az információt a fordító megjezi, és ezalapján kiválaszt egy speciális feltételes elágazást végrehajtó utasítást.

4.5.3 Működésen kívüli Végrehajtás és Regiszter Átnevezés

A legújabb CPU-k mindegyike csővezetékes (pipelined) és szuperskalár (2-6. ábra). Ez tulajdonképpen azt jelenti, hogy van bennük egy olyan egység, ami kiveszi az utasítások szavait a memóriából még mielőtt rájuk kerülne a sor, és átadja a dekódoló egységnek. A dekódoló egység azután a visszaalakított utasítást továbbítja a megfelelő működésbeli egységnek, ahol azok végrehajtódnak. Néhány esetben az utasítást felbonthatja úgynevezett mikroutasításokra attól függően, hogy az adott egység mit képes elvégezni.

A géptervezés akkor a legegyszerűbb, ha minden utasítást az előre megadott sorrendben hajtunk végre (és ha feltesszük azt, hogy az elágazáselőrejelző algoritmus sohasem téved). De ekkor a végrehajtás nem mindig optimális teljesítményű (az utasítások közötii függőségek miatt). Ha egy utasításnak szügsége van egy előző utasítás által kiszámolt értékre, a második utasításnak várnia kell, amíg az első be nem

fejeződik (Ez a RAW függőség). Mint ahogyan azt hamarosan látni fogjuk, másmilyen függőségek is léteznek.

Hogy elkerüljük ezt a problémát és hogy jobb teljesítményt érjünk el, néhány CPU átugordja a függő utasításokat egészen addig, amíg egymástól független utasításokan nem talál. Természetesen a beépített utasításütemező algoritmusnak olyan látszatot kell kelteni, mintha az utasításokat eredeti sorrendjükben hajtottuk volna végre. Egy részletes pélán keresztül illusztráljuk, hogyan működik az utasításújraszervezés.

Hogy bemutathassuk a probléma természetét, először egy olyan gépet vizsgálunk meg, ami az utasításokat a megadott sorrendben hajtja végre és értékeli ki.

A példában szereplő gép nyolc regisztert tesz elérhetővé a programozó számára R0-tól R7-ig. Minden aritmetikai művelet három regisztert igényel: kettőt az operandusoknak és egyet az eredménynek, hasonlóképpen, mint a Mic-4. Tegyük fel, azt, hogyha az utasítást az *n*. órajelciklusban dekódoltuk, akkor annak végrehajtása az *n*+0. ciklusban elkezdődik. Egyszerűbb műveleteknél -mint amilyen az összeadás vagy a kivonás- a végrehajtás az *n*+2., összetettebeknél (pl. szorzás) az *n*+3. ciklusban fejeződik be a végrehajtás. Hogy valószerű legyen a példánk, tegyük fel, hogy egy ciklus alatt két utasítást tudunk dekódolni. A kereskedelemben kapható szuperskalár CPU-k gyakran 4 vagy akár hat utasítást is dekódolnak.

A példában szereplő műveletek sora a 4-43. ábrán látható. Itt az első oszlopban az órajelciklus száma, a másodikban a művelet száma látható. A harmadik oszlop a dekódolt utasításokat tartalmazza. A negyedikben láthatjuk, hogy mejik utasítások végrehajtása kezdődik el (maximum kettő ciklusonként). Az ötödikből olvasható ki a befejezett művelet szama. Emlékezzünk rá, hogy a példánkban a kezdeti sorrendnek megfelelő bemenetre és kimenetre van szükség, azaz a *k*+1. művelet nem kezdhető el addíg, amíg a *k*. el nem kezdődött, és be sem fejezhetjük addig (nem írhatjuk az eredményét a célregiszterbe), amíg a *k*.-at nem hajtottuk végre. A többi 16 oszlop magyarázata alább található.

Registers being read: olvasandó regiszterek
Registers being written: kiírandó regiszterek
Cy: órajelciklus
Decoded: dekódolt utasítás
Iss: továbbítva
Ret: végrehajtva

4-43. ábra: Egy szuperskalár CPU működése megadott sorrendben történő utasítástovábbítással és végrehajtással.

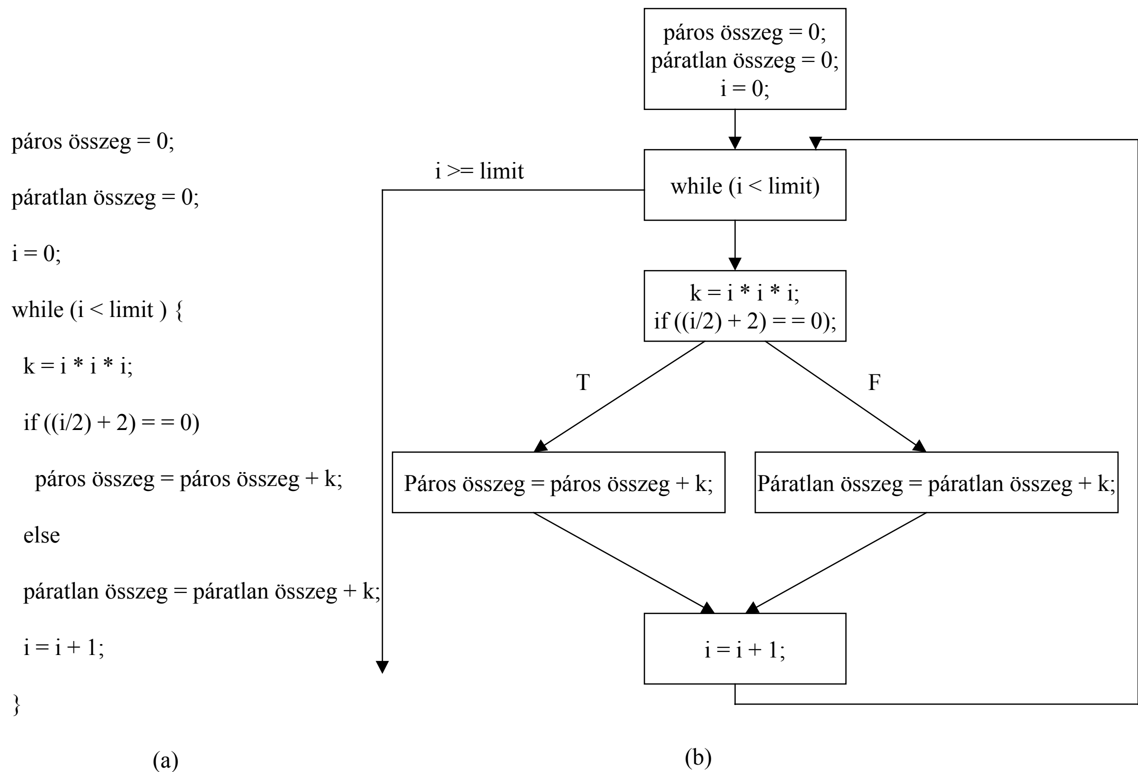
Miután dekódoltunk egy utasítást, a dekódoló egységnek el kell döntenie, hogy az adott műveletet azonnal továbbítható-e, vagy sem. Hogy ezt a döntést meg tudja hozni, ismernie kell a regiszterek állapotát. Ha az aktuális műveletnek szügsége van egy regiszterre, aminek az eredménye még nem lett kiszámolva, az utasítást nem lehet továbbítani és a CPU-nak állnia kell.

A regiszterek állapotát egy **scoreboard** nevű eszközzel követhetjük nyomon, ami először a CDC 6600-ban jelent meg. A scoreboard rendelkezik egy kis számlálóval...

***[278-281]

Nem érkezett meg! Mazán Róbert: h938349

***[282-285]



4-45.ábra (a) program részlet, (b) az ennek megfelelő blokkvázlat

a tároló és a memória közti csatornán ás vonalon elérhető kimeneti jelek, így még akár megcsinálni is megéri. Az előtte levő végrehajtó parancs ismert, ha pedig szükséges a használata, elméleti végrehajtóról beszélünk. A módszer használatához a fordítóprogram, a hardware, valamint a szerkezeti terjedelmek teherbírásának ismerete szükséges. A legtöbb esetben az alapváz határain újra ismétlődő utasítások a hardware képességeit meghaladják, ezért a fordítóprogram a parancsokat félreértés kizáróan kell, hogy továbbítsa.

Az elméleti végrehajtás néhány érdekes problémát vezet be. Először is, alapvető, hogy egyetlen elméleti parancsnak se legyen megváltoztathatatlan eredménye, mert kiderülhet, hogy esetleg nem is kellett volna őket végrehajtani. A páros és páratlan összeg kihozása /4-45.ábra/ helyes, ugyanúgy az összeadás is, amíg K alkalmazható /még az if –es állítás előtt/, de nem

jó az eredményeket a memóriában tárolni. Bonyolultabb kódszámsorokban, általános módszer az elméleti kód által használt célregiszterek átnevezése annak megelőzésére, hogy az elméleti kód felülírja a regisztereket, mielőtt még azok ismerté válnának. Ebben az esetben csak a másodosztályú regiszterek módosulnak, így nem probléma, ha a kódra végül is nincs szükség. Ha kódra szükség van, akkor a másodfokú regiszter átmásolódik a helyes célregiszterre. Ahogy elképzelhető, mindezek szem előtt tartása nem egyszerű, de kivitelezhető elég adottságokkal rendelkező hardwarerel.

Az elméleti kód azonban egy másik problémát is felvet, amely nem oldható meg egyszerűen regiszterátnevezéssel. Mi történik akkor, ha egy elméletileg végrehajtott parancs kivételt képez? Kellemetlen de nem végzetes probléma, példa erre a LOAD parancs, amely nagy tároló kapacitású /mondjuk 256 byteos/ gépeken, valamint a CPU és cachenél lassabb memóriájúakon hibát okoz. Bárhogy is, eredményesség ellen vall az, hogy ahhoz hogy egy szó révbe érjen, leállítja az épp futó programot, s ráadásul kiderül róla, hogy nem is szükséges. Túl sok ilyen optimalizáció a CPUt

még annál is lassabbá teheti, mintha az optimalizációt végre se hajtotta volna. (Ha gépnek virtuális memóriája van, ezt a 6. fejezetben tárgyaljuk, az elméleti LOAD még laphibásodást is okozhat, amely lemezvezérlést igényel, hogy beállítsa a szükséges lapot. A rossz laphibák szörnyű hatással lehetnek a teljesítőképességre, ezért fontos kiküszöbölni őket.) Számos modern gépben megoldás lehet, ha rendelkezünk egy speciális elméleti LOAD paranccsal, amely megpróbálja a szót a tárolóból a rendeltetési helyére vinni, de ha az nincs jelen, akkor megválik tőle. Ha az érték ott van amikor éppen szükséges, akkor felhasználható, de ha nincs ott, akkor a hardwarenek azonnal meg kell szereznie azt. Ha az érték nem szükséges akkor sem fizetünk rá a tárolási hibára.
Egy szerencsétlenebb esetet a következő állítással mutatunk be:

```
if (x>0)z=y/x;
```

ahol x,y és z flooting-point változó. Tegyük fel, hogy a változók előre be vannak töltve a regiszterbe és a (lassú) flooting-point megosztás eleget tesz az if es feltételnek. Sajnos az x 0 lesz, így a csapda (0-val való osztás) megszünteti a programot. A hálózati eredmény az, hogy az elmélet egy helyes program meghíusulását okozta. Még rosszabb az, hogy a programozó egyértelmű kódot iktatott be az eset elkerülésére és az így is megtörtént. Ez a helyzet nem valószínű, hogy a programozót boldoggá teszi.

Egy lehetséges megoldás lehet, ha rendelkezünk speciális változatú parancsokkal, amelyek kivételeket képezhetnek. Ezen felül

minden regiszterhez adott egy bit, ezt poison (méreg) bitnek nevezik. Ha egy speciális elméleti parancs nem ér célt, ahelyett, hogy csapdába csalná a programot, beveti a poison bitet az eredményregiszterre Ez a módosítás csak akkor derül ki, ha ezt a regisztert később egy általános információ éri. Ha az eredmény regiszter tartalmát később egyáltalán nem használjuk fel, a poison bit kitörlődik és nem okoz problémát.

4.6 A mikrofelépítésű szint jellemzői

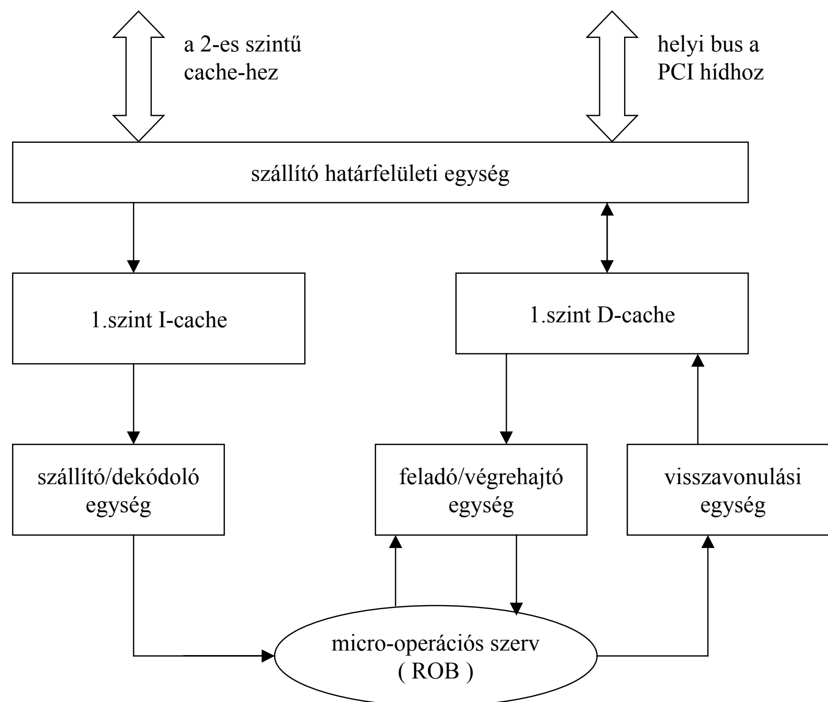
Ebben a részben 3 processzor állapotrajzain keresztül mutatjuk be röviden azok jellemzőit, rámutatva arra, hogy, hogyan alkalmazzák a fejezetekben tárgyalt fogalmakat. Mindez kényszerűségből rövid lesz, mivel a valódi gépek igen összetettek, milliónyi kapuval rendelkeznek. A példák megegyeznek az eddigiekben használtakkal: a PentiumII, az UltraSPARCII és a picoJavaII.

4.6.1 A PentiumII-es CPU mikrofelépítése

A PentiumII az Intel CPU család csúcsmodelljét testesíti meg. 32-bit operandumot és aritmetikus, valamint 64 bit flooting point működést tart fenn. Fenntart még 8 és 16 bit-es operandumokat és operációkat, amelyek a család korábbi processzorainak

hagyatéka. 64GB memóriát könyvelhet el és egyszerre 64 bit-es memória olvassa a szavakat. A 3-50. ábrán egy PentiumII-es rendszert illusztráltuk. Mint már korábba tárgyaltuk és illusztráltuk a 3-43.ábrán, a PentiumII SEC csomag két egyesített áramkörből áll: a CPU-ból (beleértve az osztott 1-es szintű tárolókat), és az egységesített 2-es szintet. A 4-46.ábra mutatja a CPU elsődleges összetevőit, úgy mint a szállító /dekódoló/, feladó /végrehajtó és visszavonuló egységek, amelyek együtt magas szintű pipeline-ként működnek. Ez a 3 egység közös parancsszervben lévő információ egy ROB (ReOrder Buffer) nevű jegyzékben van eltárolva. Röviden a szállító/dekódoló egységparancsokat szállít, és a ROB-ban való mikrooperációs tárolásra kényszeríti. A feladó/végrehajtó egység micro-operációját a ROB-tól veszi, majd megsemmisíti azt. A visszavonulási egység befejezi az összes micro oparáció megsemmisítését és naprarakész állapotba hozza, modernizálja a regisztert.

A parancsok sorba érkeznek a ROB-ba, használaton kívül kiiktathatók, de ugyancsak sorrendben vonulnak vissza.



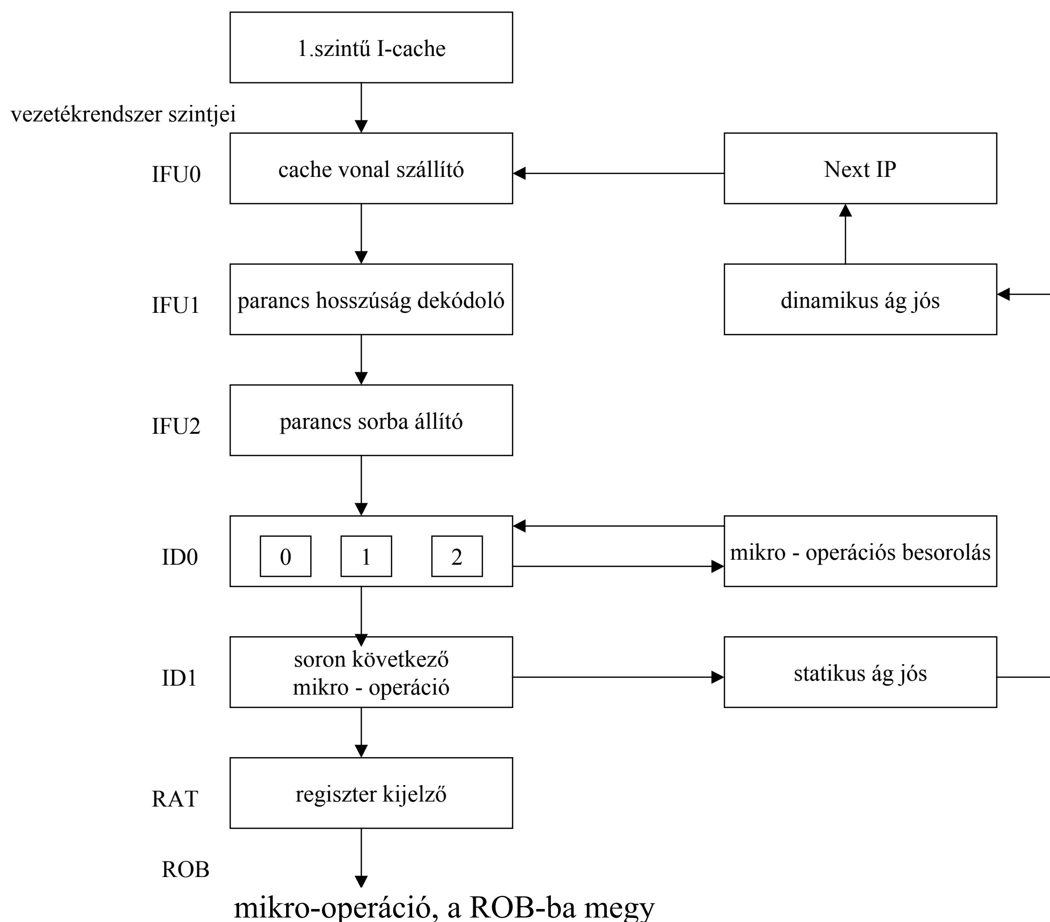
4-46.ábra A Pentium II mikrofelépítése

A szállító határfelületi egység a memóriarendszerrel, a 2-es szintű tárolóval és a fő memóriával való kommunikációért felelős. A 2-es szintű tároló nincs összekötve a helyi busszal, így a szállító határfelületi egység dolga az adatszállítás a főmemóriából a helyi buszon keresztül és a tárolók feltöltése.A PentiumII, a MESI tároló összefüggés protokolját használja, amit a 8.3.2-es egységhez, a multiprocesszorokhoz érve ecsetelünk.

A szállító/dekódoló egység

Ez az egység magasszintű vezetékrendszer, a 4-47.ábrán IFU0-tól a

ROB-ig feliratozva (A feladó/végrehajtó és visszavonuló egységeknek még külön 5-12 szintes vezetékrendszere van.) A parancsok az IFU0 szinten lépnek be a vezetékrendszerbe, ahol 32 byte-nyi vonal törlődik az I tárolóról. Valahányszor a belső puffer kiürül, egy másik tárolóvonal másolódik be. A NEXT IP regiszter irányítja a szállítófolyamatot.



4-47.ábra A szállító/dekódoló egység (egyszerűsített) belső felépítése

Mivel a gyakran IA-32-nek nevezett, Intel instrukciós szettnek változó hosszúságú parancsai vannak, számos formátumon a következő vezetékszakasz, az IFU1, ellenőrzi a byte folyamatot, hogy minden parancs kezdetét helyére tegye. Ha szükséges az IFU1 több mint 30db IA-32 parancsnak előre utána tud nézni. Az ilyen előre caló utánanézés sajnos 4 vagy 5 feltételes ágat szembeállít egymással, mindezeket pontosan előre megjelölni nem lehet, tehát nincs sok értelme ennyire előbbre vizsgálódni. A IFU2 szakasz sorba állítja az instrukciókat, így a következő

szakasz könnyen dekódolhatja azokat.

***[286-289]

286-289 oldalak

A dekódolás az ID0 fázisban kezdődik. A dekódolás a Pentium II.-nél azt jelenti, hogy minden IA-32-es utasítást átkonvertál egy vagy több mikrokóddá, éppen úgy, mint azt a Mic-4-es tette. Az egyszerűbb IA-32-es utasítások, mint például a regiszterből regiszterbe való mozgatások, egyetlen mikrokóddá fordítható. Az összetettebbek akár négy mikokód hosszúak is lehetnek. Néhány különösen összetett IA-32 utasítás még négynél is több mikrokóddá fordítódik le és a Mikrokód Sekvencer ROM segítségével generálódik a mikrokódok helyes sorrendje.

Az ID0 fázisnak három belső dekódere van. Ezek közül kettő az egyszerűbb utasításokra specializálódott, a harmadik kezeli a maradékot. Ami kijön az ID0 fázisból az egy mikrokód sorozat. Minden mikrokód tartalmaz egy műveleti kódot, két forrásregisztert és egy célregisztert.

Az ID1 fázisban sorba állítódnak a mikrokódok, a 4-35-ös ábrán látottakhoz hasonlóan. Ez a fázis programág előrejelzést is végez. Elöször egy statikus "jóslás" hajtódik végre, szükség esetén. Az előrejelzés több tényezőtől függ, de csak azokat veszi amelyek következnek, amelyeken már túlhaladott, azokat nem veszi figyelembe. Ezután következik egy dinamikus elágazás előrejelzés, mely egy előző utasításokon alapuló algoritmust használ, az elágazások előrejelzéséhez, mint azt a 4-42-es ábra mutatja. Az újabb technológia annyiban különbözik csak, hogy a 2 "történeti" bites helyett 4 biteset használ. Tisztában kell azzal lennünk, egy 12 részből álló pipeline esetén óriási veszteséget okoz egy hibás előrejelzés, ezért használnak ilyen 4 "történeti" bitet. Ha az elágazás nincs benne ebben az emlékeztető táblázatban, akkor a statikus előrejelzést használják.

Hogy elkerüljük a WAR és a WAW függőségeket a Pentium II.-es a regiszterátnevezést támogatja, amint ezt a 4-42-es ábrán láthattuk. Az IA-32-es parancsokban megnevezett aktuális regisztereket a mikrokódokban helyettesíthetjük a 40 belső regiszter bármelyikével., melyek a ROB-ban találhatók. Ez az átnevezés a RAT fázisban történik.

Végül egy ütemciklus alatt három darab mikrokód helyeződik át a ROB-ba. Az operendusok szintén itt gyűlnek össze, ha elérhetők. Ha a mikrokódhoz tartozó operandusok és az eredményregiszter elérhetők és a végrehajtó egység szabad, akkor végrehajtódik a mikrokód. Különben a ROB-ban marad , amíg minden erőforrás fel nem szabadul.

A funkcionális/végrehajtó egység

Most elérkeztünk az funkcionális/végrehajtó egységhez,melyet a 4-48-as ábra ilusztrál.Ugyanez az egység ütemezi és végrehajtja a mikrokódokat, feloldva a függőségeket és az erőforrás ütközéseket. Bár csak három ISA utasítás dekódolható egy ütemciklus alatt (ID0-ban), és legfeljebb öt mikrokód adható át a végrehajtó egységnek, minden porton egy mikrokód. Ez az arány nem tartható fenn, mivel meghaladja a törlőegység kapacitását. A mikrokódok nem sorrendben is megadhatók, de a visszarendező egység a helyes sorrendbe rakja ezeket. Egy összetett eredményjelző táblát használnak a függőben lévő mikrokódok , regiszterek és

végrehajtó egység rendszerezésére. Amikor egy mikrokód végrehajtásra kész , végrehajtódhat, még akkor is , ha egy másik, amelyet korábban tettek a ROB-ba még nem áll készen a végrehajtásra. Amikor egy többszörös mikrokód vár a végrehajtásra ugyanennél a végrehajtó egységnél, akkor egy komplex algoritmus kiválaszja , hogy melyik legyen a következő. Például egy elágazás végrehajtása sokkal fontosabb , mint egy aritmetikus uatsítás elvégzése, mivel az előbbi befolyásolja a program menetét.

4-48-as ábra: Az funkcionális/végrehajtó egység.

Ez a funkcionális/végrehajtó egység tartalmaz egy várakozó állomást és végrehajtó egységeket, melyek 5 porttal csatlakoznak. A várakozó állomás egy 20 mikrokód kapacitású tároló olyan mikrokódok számára, melyeknek minden operandusuk megvan. Ezek a várakozóállomáson várnak a sorukra, míg a szükséges végrehajtól egység szabad nem lesz.

A várakozóállomás 5 porttal csatlakozik a végrehajtó egységekhez. Néhány egységnek közös portja van , amint ez az ábrán is látható. A betöltő (Load)- és a tároló (Store) egységek hajtják végre a megfelelő információt. Két port van a raktározás számára. Mivel csak egy mikrokód bocsátható át ciklusonként és portonként, ezért ha két független mikrokódnak kell ugyanazon a porton keresztülmennie egyiküknek várnia kell.

A megszüntető(Retire) egység

Ha egy mikrokódot végrehajtották visszakerül a várakozóállomásra, azután a ROB-ba, ahol idővel megszűnik. A megszüntető egység felelős azért, hogy elküldje az eredményeket a megfelelő helyre -a megfelelő regiszterbe- de más állomásoknak is , melyek az funkcionális/végrehajtó egységben találhatók és várnak a megfelelő értékre. A megszerző/dekódoló egység (Fetch/Decode) a "hivatalos" regisztereket tartja számon,amelynek értékeit a már végrehajtott utsítások határozzák meg. A megszüntető egység azokat a független regisztereket tartalmazza, azaz azokat az értékeket, melyek az utasítás végrehajtása során részeredmények, mivel előző parancsok még nem hajtódtak végre.

A Pentium II. támogatja a spekulatív végrehajtást, így néhány utasjtást feleslegesen végez el és nem von vissza. Tulajdonképpen ez a visszagördítő képesség. Ha úgy dönt, hogy néhány mikrokód egy olyan IA-32 parancstól jött, melyet nem kellett volna elvégezni, az eredményeit mellőzi. A megszüntető egységtől függ, hogy készít-e feljegyzést erről. Csak "hivatalosan" elvégzett parancsokat lehet visszavonni , ennek sorrendben kell történnie , még akkor is , ha nem sorrenben kerültek elvégzésre.

4.6.2. Az UltraSPARC-II CPU mikroarhitektúrája

Az UltraSPARC sorozatok a 9-es verziójú SPARC arhitektúra Sun implementációja. A különböző modellek viszonylag hasonkóak, főleg teljesítményben és árban különböznek. Az összetévesztés elkerülése végett ebben a részben az UltraSPARC II.-re fogunk utalni és ennek tulajdonságait írjuk le azokon a területeken, amelyeken különböznek.

Az UltraSPARC II. egy teljes 64 bites gép, 64 bites regiszterekkel és egy 64 bites adat BUS-szal, bár annak érdekében, hogy a 8.-as verziójú (32 bites) SPARC-okkal lefelé is kompatibilis maradjon 32 bites operandusokat is tud kezelni és módosítás nélküli futtat 32 bites SPARC softwereket. Bár a belső arhitektúra 64 bites, a memória 128 bit széles (hasonlóan a Pentium II.-höz, melynek 32 bites arhitektúrája és 64 bites memória BUS-a van), de egy generációval a Pentium II előtt van . Az UltraSPARC II. renszer magját a 3-47-es ábra mutatja.

A teljes SPARC sorozat tisztán RISC filozófia szerint lett tervezve a kezdetektől fogva. A legtöbb utasításnak 2 forrásregisztere van és egy célregisztere, így alkalmasak az egy ciklusos pipelinera. Nem szükséges a régi CISC parancsokat RISC mikrokódokra fordítani, amint azt a Pentium II.-nek meg kell tennie.

A UltraSPARC II. egy szuperskaláris arhitektúra, mely 4 utasítást tud végrehajtani egy óraciklusban. A parancsokat sorrenben kapja, de lehet , hogy nem sorrenben hajtja végre, Viszont a megszakítás esetén nincs kétértelműség azt illetőleg, hogy a gép hol tartott , amikor a megszakítás történt.A PREFETCH parancs formájában egy hardware támogatja a spekulatív betöltéseket, ez a cache hiány miatt nem okoz hibát. Valójában még csak nem is blokkolja a következő memóriabemeneteket. Ezért a fordító előretervezhet egy vagy több PREFETCH-t a kódsztreambe, jóval előbb, minthogy szükség lenne rájuk, abban a reményben , hogy meglegyenek az adatok ott, akkor mikor kellenek, de anélkül, hogy bármilyen hátrány számazna abból, ha az adat a cacheben nem található.

Az UltraSPARC II áttekintése

Az UltraSPARC II. blokkdiagramm látható a 4-49-es ábrán. Minden látható komponens a CPU chippen van , kikéve a 2. szintű cachet , mely teljesen külsődleges. A I-cache egy 16 KB-os kétirányú asszociatív cache , 32 byteos sorral, de megvan az a lehetőség, hogy egy cachesor felét is ki tudjon olvasni. Egy fél cachesor(16 byte) pontosan 4 utasítást tartalmaz, amelyeket egy óraciklus alatt tudkiolvasni. A D-cash egy 16 KB-os direkt címzésű visszaírható, nem fix felosztású cache, amely használható 32 bit/sor módban, illetve egy sor felosztható két 16 bytos alsorra.

4-49. Az UltraSPARC II. mikroarhitektúrája

A külső cash egység kezeli az 1-es szintű cachehiányokat (azt jelenti, hogy az 1ües szintű cacheben nem található meg a szükséges adat) , úgy, hogy a szükséges sort megkeresi a külső (azaz 2-es szintű) cacheben. Ha a keresésés sikeres a sort bemásolja a megfelelő 1-es szintű cashbe. Ha a keresés sikertelen a külső cache egység megkéri a memória illesztőegységtől a szükséges adatot a főmemóriából.

***[290-293]

Most tekintsük az Ultra SPARC II belső funkcionális egységeit.A Prefetch / Dispatch (előreolvasó / elküldő) egység nagyjából hasonló a Pentium II-ben a Fetch / Decode (beolvasó / dekódoló) egységhez (lásd 4-46 ábra). Azonban a feladata egyszerűbb, mert a beérkező utasítások már három regiszteres mikrokódok, tehát nincs arra szükség, hogy lefordítsák őket. Időveszteség nélkül az egység vagy az 1. szintű cache-tárából vagy a 2. szint cache-tárából olvashat be. Négy utasítás olvasható be óraciklusonként.

Az elágazás hatásának csökkentéséhez, az 1. cache-tárban minden csoportnak (egy csoport négy utasításból áll) van egy cime, amely megmondja, melyik csoportot használják a legközelebb. Továbbá, a Prefetch / Dispatch egységen belül egy teljes elágazás előrejelző van elhelyezve. Ez egy 2-bites előrejelzési algoritmust használ, amely hasonló a 4-42 ábrán látható algoritmushoz. Továbbá, az UltraSPARC II-nek van egy elágazási utasításkészlete, amelyben a fordító meg tudja mondani a hardvernek milyen módon jelezzen előre, mivel gyakran saját magának elég jó ötlete van. Az előrebeolvasott utasítások egy 12 elemű várakozási sorba kerülnek, a Grouping (csoportosítási) logika szerint.

A Grouping logikai egység négy utasítást választ ki a várakozási sorból a kibocsátáshoz. A trükk az, hogy négy olyan utasítást kell találni, amelyeket egyszerre lehet kibocsátani. Az Integer (egész) végrehajtó egységnek két különálló ALU-ja (aritmetikai és logikai egysége) van, ahhoz, hogy két utasítást párhuzamosan tudjon végrehajtani. Hasonlóan a Floating-point (lebegőpontos) egységnek szintén két FP ALU-ja van. Következésképpen, egy csoport tartalmazhat minden típusból kettőt, de négyet egyik típusból sem. Hogy a csoportosítás optimális legyen, az utasításokat nem az eredeti sorrendben kell kibocsátani, amely megengedhető. Ezek szintén nem az eredeti sorrendben lesznek visszavonva.

Mind az Integer, mind a Floating-point egység tartalmazza a megfelelő regisztereket, tehát az utasítások a megfelelő operandusaikat szintén az egységen belül találják és az eredmények szintén ide kerülnek. Az Integer és Floating-point regiszterek különállóak, tehát az értékekik sosem másolhatók át egyik egységből a másikba. A Floating-point egység egy Grafikai egységet is tartalmaz, amely a 2D és 3D grafikához, audio és video alkalmazásokhoz speciális utasításokat hajt végre, a PentiumII MMX utasításkésszletéhez hasonlóan.

A Load/Store (betöltő / tároló) egység betöltő / tároló utasításokat kezel. Ha a szükséges adat az 1. szintű D-cache-ben található, akkor késedelem nélkül hajtódik végre. Máskülönben az External (külső) cache egység a 2. szintű cache-ből vagy a memóriából olvassa be vagy írja ki az adatot. Ha a D-cache-tárat írási üzemmódban használták és egy tárolási memóriahiány lépet fel, a szükséges cache-tár vonalat a 2-es szintű cache-ből vagy a fő memóriából hozták volna be. Az egyedüli szót, amely tárolva van, éppen a 2-es szintű cache-be van beírva, vagy ha az szintén telített, a fő memóriába. Hogy elkerüljük a D-cache hiányok halmozódását, a Load/Store egység függő betöltési és tárolási várakoztatási sorokat tart fenn, így új utasításokat tud feldolgozni az alatt, amíg arra vár, hogy a régiek befejeződjenek.

Az UltraSPARC II Pipeline-ja

Az Ultra SPARC II-nek 9 szakaszból álló pipeline-ja van. Ezekből a szakaszokból néhány eltérő az integer és a floating-point utasításokban, ahogyan a 4-

50. ábrán látható. Az első szakasz beolvassa az utasításokat az I-cache-tárból (ha lehetséges).Szerencsés esetben (nem lép fel cache hiba, nincs hibás előrejelzés, nem boyolult utasítás, a helyes utasítás párosítás, stb.), ciklusonként négy utasítás beolvasását és kibocsátását tudja korlátlanul végezni. A Decode (dekódoló) szakasz néhány többlet bitet ad minden utasításhoz, mielőtt áthelyezi azt a fent említett 12-elmű várakozási sorba. Ezek a bitek az újabb feldolgozást gyorsítják fel(például azáltal, hogy azt rögtön a megfelelő végrehajtási egységbe irányítja).

4-50. ábra. Az UltraSPARC II pipeline-ja

A group szakasz megfelel a Group logikának, amelyet már korábban láttunk. Ez a dekódolt utasításokat rendezi négyes csoportokba, amelyek egyidejű végrehajtásra alkalmasak.

Ezen a ponton az integer és a floating-point pipeline-ok szétválnak. A negyedik szakasz az integer egységben a legtöbb utasítást egy ütemciklus alatt hajtja végre. Azonban, a LOAD és STORE utasításoknak külön feldolgozásra van szükségük a cache szakaszban. Az N1 és N2 szakaszok nem csinálnak semmit a regiszteres utasításokkal, feladatuk az, hogy a két csatornát összhangban tartsák. Mindegyik integer utasítás néhány nanoszekundum alatt befejeződik , azonban kis időveszteséggel jár, hogy a pipeline folyását egyenletesen fenn lehessen tartani.

A floating-point egységnek négy szakasza van. Az első szakaszban tudunk hozzáférni a float-point regiszterekhez. A következő három (szakasz) az utasítás végrehajtásához szükséges. Minden floating-point utasítás három óraciklus alatt hajtódik végre, kivéve az osztást (12 ciklus) és a gyökvonást (22 ciklus), tehát más floating-point utasítás nem lassítja le a pipeline-t.

Az N3 szakaszban , mely mindkét egység számára közös , olyan megszakításokat kezelnek amely a végrehajtás során előfordulhatott, mint pl. a zéróval való osztás. Végül ,az utolsó szakaszon, az eredmények visszaíródnak a regiszterekbe. Ez a szakasz összehasonlítható a PentiumII Retite (visszavonó) egységével, azért, mert, ha az utasitás átment ezen a szakaszon, akkor azt végrehajtják.

4.6.3 A picoJava II CPU mikroarchitektúrája

A picoJava II JVM bináris programokat tud futtatni szoftver interpretáció nélkül. A legtöbb JVM utasítást a hardver közvetlenül egy óraciklus alatt hajtja végre. Körülbelül 30 JVM utasítást mikrokódoltak. A picoJava II hardver csak nagyon kicsi számú utasítást nem tud végrehajtani, és ezeket szoftveresen kell emulálni. Ezek a utasítások tipikusan a JVM olyan sajátságaival foglalkoznak, amelyekről nem beszéltünk, úgy mint a komplex szoftver objektumok alkotása és kezelése.

A picoJava II áttekintése

A picoJava mikroarchitektúra blokkdiagramja a 4-51. ábrán látható. A chip 1. szintű cache-táron egy szakadás van. Az I-cache-tár opcionális és a következő

értékeket veheti fel: 0 KB, 1KB, 2KB, 4KB, 8KB vagy 16KB. Ez egy 16 byte vonalszélességű közvetlenül címzett tár. A D-cache szintén opcionális, a következő értékeket veheti fel:0 KB, 1KB, 2KB, 4KB, 8KB vagy 16KB. Ez egy kétirányú asszociatív cache, 16 byte-os vonalszélességgel. Ez writeback-et és allokálást (lefoglalást) használ. Mindegyik cache-tárnak 32 bites csatlakozója van a memóriához. A microJava 701-be mindkét cache be van építve, mindegyik a maximális 16 KB-al. A picoJava II-nek a floating-point egység is opcionális része.

4-51. ábra A picoJava blokkdiagramja mindkét 1. szintű cache-tár és lebegőpontos egységgel. Ez a microJava 701 konfigurációja.

Az I-cache-tár egy ütemciklus alatt 8 byte-al látja el a Prefetch/Decode, folding(gyűjtő) rendszert. Ez az egység a következő körben a végrehajtási kontrollerhez és a fő adat bushoz (integer/floating-point egység) van kapcsolva. Az integer műveletek számára az adatbus 32-bit széles. Ez kezelni tudja a szimpla és dupla pontosságú IEEE 754 lebegőpontos számokat is.

A 4-51 ábrán a legérdekesebb része egy regiszterfájl, amely 64 32-bites regiszterből áll. Ezek a regiszterek a JVM vermének a legfelső 64 szavát tárolhatják, nagyon felgyorsítván a veremhozzáféréseket. Mindkettő, tehát a műveletvégző és a lokális változó verem, tárolható a regiszterfáljban. A regiszterfálj elérése szabad (tehát minden időbeli késleltetés nélküli, amíg D-cache elérése külön időt igényel). A sín az integer és a floating-point egységek között 96 bit széles, és végrehajtható egy ütemciklus alatt egy 32 bites olvasás a veremből egy 32 bites veremírással együtt.

Ha az operandus verem két szót tartalmaz, akkor lokális változó verem legfeljebb 62 szót tartalmazhat a regiszterfáljban.Természetesen, hogyha egy következő szót is a veremben tárolunk, probléma lép fel. Az történik ugyanis, hogy a verem alján lévő egy vagy több szó ki van véve a veremtároló tetejéről, hely keletkezik a regiszterfáljban, így a veremtárolóban néhány mélyen levő szót újra be lehet tölteni a regiszterfáljba. Speciális regiszterek a chipben meghatározzák azt, hogy a regiszterfáljnak milyen tele kell lennie, mielőtt néhány, a verem alján lévő szót a memóriába írnak és azt, hogy milyen üresnek kell lennie mielőtt a regiszterfájlt újra feltöltik a memóriából. A dribbling megkönnyítéséhez, hogy másolást ne kelljen végrehajtani, a regiszterfájl úgy működik, mint egy körkörös tároló, két pointerrel, amely a legalacsonyabb és a legmagasabb érvényes szóra mutat. A dribblingezés automatikus, valahányszor a regiszterfájl nagyon megtelik, vagy nagyon megürül.

A picoJava II pipeline-ja

A picoJava II-nek egy hat szakaszos pipeline-ja van. Ez a 4-52. ábrán látható. Az első utasítást tölt be az I-cache-ből az utasítástárlóba, melynek kapacitása 16 byte. A következő szakasz dekódolja és kombinálja az utasításokat oly módon, hohy röviden legyenek leírva. Ami a dekódoló egységből kijön, a mikrokódok egy

sorrendje, amelyek mindegyike egy opkódot és három regiszterszámot tartalmaz (két forrásregisztert és egy célregisztert). Ebben a vonatkozásban a picoJava II hasonló a Pentium II-höz: mindkét gép egy CISC utasításáramlatot fogad el mely belsőleg egy mikrokódos sorozatra van felosztva, mely egy csatornázott RISC-be van betáplálva. Habár a Pentium II-vel ellentétben, a picoJava nem szuperskaláris és a mikrokódokat olyan sorrendben hajtják végre és vonják vissza amelyben ezeket kibocsátották. Egy cache-tár hiba esetén az operandust a főmemóriából kell behozni, a CPU megáll és várakozik.

4-52. ábra A picoJava II-nek egy 6-szakaszos pipeline-ja

A harmadik pipeline szakasz a veremtárolóbol operandusokat hoz (valójában a regiszterfájlból) úgy, hogy ezek készen lesznek a negyedik szakaszhoz, a végrehajtó egységhez, ahol...

***[294-297]

Nem érkezett meg!

***[298-301]

Nem érkezett meg!

***[302-305]

29. Egy 5 szintű csővezetékes számítógép feltételes ággal dolgozik azáltal, hogy akadályozza a következő három ciklust, miután egyet megtett. Az akadályozás mennyire érinti a teljesítményt, ha az utasítások 20%-a feltételes ág? Hagyjon figyelmen kívül minden akadály forrását, kivéve a feltételes ágakét.

30. Tételezzük fel, hogy a számítógép több, mint 20 utasítást hoz elő. Bárhogy is, átlagban 4 ezekből feltételes ág, mindegyik 90%-os valószínűséggel megjósolható. Mi a valószínűsége annak, előhozás hogy a helyes úton van?

31. Tegyük fel, hogy meg akarjuk változtatni a használatban lévő számítógépünk terveit 4-43 ábra alapján, hogy 16 regiszter legyen 8 helyett. Ezután kicseréljük a I6-t R8-ra a cél érdekében. Mi történik a ciklusban a 6. Ciklustól kezdve.

32. Általában a függőség problémákat okoz a csővezetékes CPU-kban. Vannak olyan lehetőségek, amelyeket alkalmazva a WAW függőségggel kapcsolatos problémák megoldódhatnak. Mik ezek?

33. Írja újra a Mic-1 értelmező programot, de az LV most mutasson az első helyi válozóra a kapcsolódási pont helyett.

34.Írjon egy szimulációt sz 1 utas közvetlen térképű cache-hez. Készítse el a szimuláció kimeneteinek és a vonalak méreteinek paramétereit. Vizsgálja meg és írja le a tapasztalatait.

5
A gépi utasítások szintje

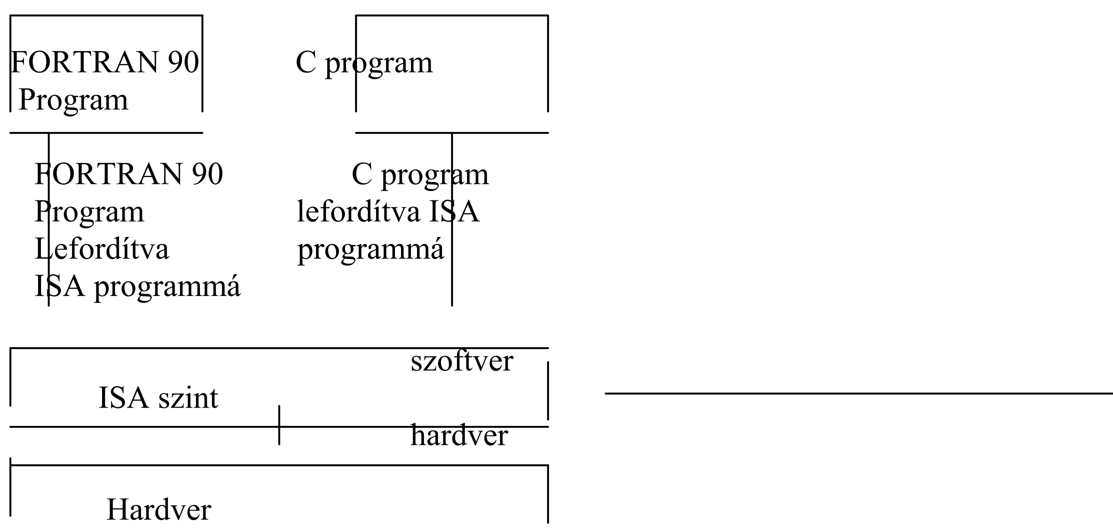
Ez a fejezet a gépi utasítások (ISA) szinttel részletesen foglalkozik.
Ez a szint a mikroprogramozás szintje és az operációs rendszerek szintje között helyezkedik el, ahogyan ez az 1-2. ábrán is látható. Történetesen ezt a szintet a többi szint előtt fejlesztették ki, és valójában ez volt az egyetlen szint. Mára nem ritka, hogy csupán gépi architektúra vagy néha (helytelenül) "assembly nyelvként" utalnak rá.

Az ISA szint különleges jelentősége az, ami fontossá teszi a rendszer-mérnökök számára : a kapcsolat a szoftver és a hardver között. Habár lehetőség van rá, hogy a hardver közvetlenül futtasson C, C++, FORTRAN 90 vagy más magas szintű nyelven írt programokat, de ez nem tanácsos. Továbbá, hogy praktikusabb legyen a használata, a legtöbb számítógépnek

képesnek kell lennie, hogy olyan programokat futtasson, amelyek összetett nyelven íródtak, nem csak egyetlenen.

Az a megközelítés, hogy alapvetően minden rendszertervező megszokta, hogy a különböző magas szintű nyelven ír programot lefordítják egy általános köztes formára - az ISA szitre - és építenek egy hardvert, amely képes közvetlenül futtatni az ISA szitű programokat. Az ISA szint meghatározza a kapcsolatot a fordító, és a hardver között. Ez az a nyelv, amelyet mind a kettő megért. A kapcsolat a fordító és a hardver között, ami az 5-1 ábrán látható.

Ideálisan, amikor új gépet terveznek, a mérnökök a fordítókkal is beszélnek, hogy megtudják milyen tulajdonságok kellenek az ISA szinthez.



5-1 ábra: Az ISA szint a kapcsolat a fordítók és a hardver között.

Ha a fordítók olyan tulajdonságot akarnak, amelyeket nem tudnak olcsón kivitelezni(pl.: ág-és-számlázz-utasítás), akkor abba nem mennek bele.

Egyszerűen, a hardver-barátok ha van egy új, remek tulajdonság, amit bele akarnak rakni (pl.: memória, ami szó szerint szuper gyors a címek és prímszámok terén), de a szoftver-barátok nem tudják kitalálni hogyan írjanak kódot a használatára, nem jut tovább a tevezősztalnál. Sok tárgyalás és szimuláció után, egy ISA teljesen tökéletes a használni szándékozott programozási nyelvhez,akkor be fogják fejezni és megvalósítják.

Ez az elmélet. Most itt a zord valóság. Amikor egy gép a forgalomba kerül, az első kérdés amit a potenciális vásárlók feltesznek : "Kompatibilis-e az elődjével?" A második : "Fut-e a régi operációs rendszer rajta?" A harmadik : "Futnak-e majd a meglévő programok rajta?" Ha bármelyik válasz "nem", a tervezőknek sok megmagyarázni valójuk lesz. A vásárlók ritkán örülnek annak, ha ki kell dobniuk a régi szoftvereiket, és mindent előről kell kezdeniük.

Emiatt a felfogás miatt nagy nyomás nehezedik a számítógép-mérnökökre, hogy megtartsák az ISA-t egyformának a modellek között, vagy legalább visszafelé kompatibilissé tegyék. Ez alatt azt értjük, hogy az új gépnek változatlanul futtatnia kell a régi programokat. Bárhogy is, az teljesen elfogadható, hogy az új gépnek új utasításai és tulajdonságai vannak, amelyet csak az új szoftverek tudnak kihasználni.

Az 5-1 ábra szerint, amíg a tervezők az előző modellekkel visszfele kompatibilisé teszik az ISA-t, addig sokkal nagyobb szabadságot élveznek, mert bármit csinálhatnak a hardverrel, amit akarnak, mivel nem valószínű, hogy valaki is igazán foglalkozna a hardverrel (vagy egyáltalán tudná mit csinál). Ők váltani tudnak a mikroprogramozott tervezésről a közvetlen futtatásra, vagy hozzáadnak csővonalakat, szuperskaláris lehetőségeket vagy valami mást, amit akarnak, feltéve, hogy támogatják a kompatibilitást visszafele az előző ISA-val. A cél az, hogy meggyőződjenek arról, hogy a régi programok futnak az új gépen. A kihívás akkor épít jobb gépeket, ha eleget tesz a visszafele való kompatibilitás kényszerének. A már említettet nem kell belefoglalni, az ISA tervezésnél nem számít. Egy jó ISA jelentős előnyökkel szemben, különösen az egyszerű számítógépes erő az árral szemben. Máskülönben egyenlő tervek különböző ISA-kal számolnak a 25%-os teljesítmény növelés érdekében. A mi véleményünk szerint csak a piaci erő követelései teszik nehézzé (de nem lehetetlenné), hogy kidobják a régi ISA-kat és újat mutassanak be. Bárhogy is, amikor ritkán megjelenik egy általános célú ISA, addíg a specializált piacon (pl.: rendszerek, vagy multimédiaprocesszorok) sokkal sűrűbben.

Mit csinál egy jó ISA? Két elsődleges feladata van. Először, egy jó ISA olyan utasítások sorozatár határozza meg, amelyeket hatékonyan lehet megvalósítani a mostani és a jövőbeli technikákkal, melyek eredménye az olcsó tervezés több generáció számára. A gyenge tervet nehezebb megvalósítani, több munkát igényel előállítani a processzort és több memóriár igényel a programok futtatása. Lassabban is futhat, mert az ISA elfedi a lehetőségeit, hogy műveletek egybeesnek, ami sokkal bonyolultabb tervezést igényel, hogy ugyanazt az eredményt érje el. A tervezés amely élvezi a technika jellemző tulajdonságait, szalmalláng lehet, amely biztosítja az egyszerű generáció olcsó megvalósítását, csak azért, hogy felülmúlja egy sokkal előbbre tekintő ISA-val.

Másodszor egy jó ISA-nak tiszta teret kell nyújtania a lefordított kód számára. A választások sorának valóságossága és teljessége fontos vonások, amelyek nem mindig jellemzőek az ISA-ra. Ezek a tulajdonságok nagyon fontosak a fordító számára, ami gondot okoz amikor a legjobban kell választani a korlátozott lehetőségek közül, különösen, ha a nyilvánvaló lehetőséget az ISA nem engedi meg. Amióta az ISA a kapcsolat a hardver és a szoftver között, eleget kell tennie a hardver tervezők kívánságainak (könnyű legyen hatékonyan megvalósítani) és eleget kell tennie a hardver tervezők kívánságainak (könnyű legyen jó kódot írni rá).

5.1. Az ISA szint áttekintése

Kezdjük azzal az ISA tanulmányozását, hogy mi is az. Ez így könnyű kérdésnek tűnik, de több bonyodalom merülhet fel, mint amit gondolnánk. A következő fejezetben felvetünk néhányat a következő témák közül. Meg fogjuk nézni a memóriák modelljeit, regisztereket és utasításokat.

5.1.1. Az ISA szint tulajdonságai

Először is, az ISA szintet az határozza meg, hogy a gép milyen a gépi nyelvet programozók számára. Nincs olyan (józan) ember, aki többé programot ír gépi nyelven, határozzuk meg újra, nevezzük az ISA szintű kódot annak, amit a fordító kiad (figyelmen kívül hagyva az operációs rendszer hívását és a szimbólikus assembly nyelvet egy pillanatra). Hogy ISA szintű kódot állítson elő, a fordító

***[306-309]

Ezen információk összessége határozza meg az ISA szintet.
Az előző definíció alapján az, hogy milyen a mikroarchitektúra (mikroprogramozott, pipeline, superscalar, és így tovább) nem része az ISA szintnek, mert a programozó nem látja ezeket. Ez az állítás azonban nem teljesen igaz, mert ezen tulajdonságok közül néhány befolyásolja a teljesítményt, és ezt már észrevehetik a programozók. Mindezeket figyelembe véve, például egy superscalar megvalósítás képes un. back-to-back utasítások végrehajtására ugyanabban a ciklusban, feltéve, hogy az egyik műveletben egész, a másikban lebegőpontos számokat használ. Ha a fordító váltogatja az egész és lebegőpontos utasításokat, akkor az észrevehetően jobb teljesítményt eredményez. Így a superscalar utasításvégrehajtás bizonyos részei észlelhetők a ISA szinten, tehát a szintek közötti határvonal nem olyan egyértelmű, mint ahogy az elsőre tűnhet.
Néhány felépítés ISA szintjéhez leírást is közzé tesznek. Például a V9 SPARC-nak (Version 9 SPARC) és a JVM-nek is van ilyen hivatalos dokumentációja (Weaver and Germond, 1994; és Lindholm-Yellin, 1997). Azért tesznek közzé ilyen információkat, hogy lehetővé tegyék más fejlesztőknek is, hogy olyan gépet építsenek, melyen ugyanazt a szoftvert futtatva pontosan ugyanazt az eredményt kapjuk.
A SPARC esetében megengedték, hogy mások is tudjanak olyan SPARC chipet gyártani, amelyek funkciójukban teljesen megegyeznek az eredetivel, különbség csak árukban és teljesítményükben legyen. Ehhez a többi fejlesztőnek tudnia kell, hogy mit is tud egy SPARC chip (az ISA szinten). A dokumentáció ezért leírja, milyen memóriamodellt használ, milyen regiszterei vannak, mit csinálnak az egyes utasítások, és így tovább, de arról nincs bennük szó, hogy mindezt hogyan is valósították meg az eredeti processzorban.
Minden ilyen dokumentáció tartalmaz un. **normative** részeket, amelyek rögzítik a követelményeket, és **tájékoztató** részeket, melyeknek az a feladata, hogy segítse az olvasót. A normative rész nem mindig egyértelműen fogalmaz azért, hogy bizonyos dolgokat eltitkoljanak. Példaként nézzük meg a következő mondatot: *Egy fenntartott opcode futtatása csapdát okozhat.* Ez azt modja, hogy ha a program egy olyan opcode-ot futtat, amely nem definiált, akkor annak hibát kell okoznia, ezt nem szabad egyszerűen figyelmen kívül hagyni és átugrani. Egy másik választási lehetőség, hogy ezt a kérdést nyitva hagyjuk. Ebben az esetben a mondat így nézne ki: *Egy foglalt opcode futtatása a megvalósításban definiált.* Ez azt jelenti, hogy a programozónak nem kell semmilyen különös viselkedésre számítania, szabadságot adva így a gyártóknak. A legtöbb architektúrát letesztelnek, hogy valóban tudja-e azokat a dolgokat, amiket elvárnak tőle.
Világos, hogy miért is van a V9 SPARC-nak ilyen dokumentációja, amely leírja az ISA szintjért: azért, hogy minden V9 SPARC chip ugyanazokat a szoftvereket tudja futtatni. Hasonló okoból van a JVM-nek is leírása: azért, hogy minden olyan processzor, mint a picoJava II futtatni tudjon legális JVM programokat. Nincsen viszont dokumentáció a Pentium II ISA szintjéről, mert az Intel nem akarja, hogy más processzorgyártó is tudjon Pentium II-t készíteni. Valóban, az Intel bírósághoz is fordult, hogy megállítsa processzorainak klónozását.
Másik fontos tulajdonsága az ISA szintnek, hogy a legtöbb gépnek legalább két módja van. A **kernel mód** arra szolgál, hogy futtassa az operációs rendszert és lehetővé tegye az utasítások végrehajtását. A **felhasználói (user) mód** feladata az

alkalmazások futtatása, de nem engedi bizonyos érzékeny utasítások végrehajtását (mint például a közvetlen beavatkozást a cache-be). Ebben a fejezetben főleg a felhasználói mód utasításaival és tulajdonságaival fogunk foglalkozni.

5.1.2 Memóriamodellek

Minden számítógép felosztja a memóriát memóriacellákra, amelyeket egymás után megcímez. Manapság egy cella 8 bites, de régebben 1-60 bites cellákat is használtak (lásd 2-10-es ábra). Egy 8 bites cellát byte-nak hívnak. Azért 8 bites rekeszeket használnak, mert az ASCII karakterek 7 bitesek, így egy ASCII karakter plusz egy paritásbit éppen egy byte-ot ad. Ha majd az UNICODE lesz az elterjedtebb, akkor talán a jövő számítógépeinek memóriája 16 bites, folyamatosan címezhető egységekre lesznek bontva. Összességében 2^4 mindjárt másképp fest, mint 2^3, mert a 4 a 2 hatványa, míg a 3 nem.
A byte-okat általában 4 byte-os (32 bit) vagy 8 byte-os (64 bit) csoportokra osztják, és hozzájuk olyan utasításokat készítenek, amelyek ilyen szavakkal képesek dolgozni. Néhány megvalósításban a szavaknak rendezve kell lenniük, így például egy 4 byte-os szónak a 0, 4 vagy 8 címen kell kezdődnie, és nem az 1 vagy 2 címen. Teljesen hasonlóan, egy 8 byte-os szónak a 0, 8 vagy 16 címen kell kezdődnie, de nem kezdődhet 4-en vagy 6-on. Egy 8 byte-os szó igazítását az 5-2-es ábra mutatja.

5-2-es ábra: Egy 8 byte-os szó egy un. little-endian memóriában. (a) Igazított. (b) Nem igazított. Néhány gép megköveteli, hogy a szavak igazítva legyenek memóriájában.

Az igazítás gyakran szükséges, mert a memóriák így sokkal hatékonyabban működnek. Például a Pentium II, amely egyszerre 8 byte-ot tölt be a memóriából, 36 bites címet használ, de ebből csak 33 bitet utal a címre ahogy ezt a 3-44-es ábrán is láthatjuk. A Pentium II még akkor sem tudna rendezetlen memóriából olvasni, ha akarna mert az alsó 3 bit gyakorlatilag nem része a memóriacímnek. Ezek mindig nullák, ezzel "kényszerítve" a memóriát, hogy a címek 8 többszörösei legyenek.
Az igazítás szükségessége néha jelentős problémákat okoz. A Pentium II-n az ISA szintű programok megengedik, hogy bármilyen című szóra hivatkozzunk. Ez a 8088-as időkbe nyúlik vissza, mert a 8088-nak 1 byte széles adatbusza volt (és így nem volt szükség igazításra). Amikor egy Pentium II-es program beolvas egy 4 byte-os szót, amely a 7-es címen kezdődik, akkor a hardvernek végre kell hajtani egy memóriaolvasást 0-tól 7-ig és egy másikat 8-tól 15-ig. Ezután a beolvasott 16 byte-ból ki kell választani a 4 szükséges byte-ot, és összeállítani belőle a helyes bytesorrendű 4 byte-os szót.
Az, hogy bármilyen címről szavakat tudjunk olvasni speciális logikát igényel a processzortól, amelyek így nagyobbá és sokkal drágábbá válnak. A processzortervező mérnökök szívesen megszabadulnának ettől, így csak a programokat kellene átalakítani úgy, hogy igazítva írjanak és olvassanak a memóriában. A probléma csak az, hogy a tervezők kérdésére: "Kit érdekelnek azok a réges-régi 8088-as programok amelyek rosszul használják a memóriát?" röviden így válaszolnak: "Az ügyfeleinket".
A legtöbb gépnek egy lineárisan címezhető területe van az ISA szinten, amely egy 0 címtől egy bizonyos felső határig címez, általában 2^{32}-ig vagy 2^{64}-ig. Néhány gépnek külön memóriaterülete van az utasítások és az adatok számára, tehát az az

utasítás, amely a 8 címen kezdődik mást jelent, mint az az adat, amely szintén 8 címen kezdődik. Ez a módszer sokkal bonyolultabb, mint ha egyszerűbb, lineárisan kezelhető memóriánk lenne, ám az előző eljárásnak van néhány előnye. Először is, ez lehetővé teszi, hogy 2^{32} byte terület legyen programjaink számára és 2^{32} byte az adatoknak is, miközben csak 32 bites címeket használunk. Másodszor, minden írás autómatikusan az adatterületre megy, elkerülve ezzel a program véletlenüli felülírását, tehát egy hibalehetőségét.
Felhívjuk a figyelmét, hogy a külön utasítás- és adatterület nem ugyanaz, mint az 1-es szintű osztott cache (split level 1 cache). Az első esetben a teljes címezhető memória mérete megduplázódott, és valamely címről való olvasás eltérő eredményt ad attól függően, hogy az utasítás- vagy az adatterületről olvastunk. Egy osztott cache-ben egyféle cím van, csak a memóriaterület egyes részeit különböző cachek tárolják.
A memóriamodellek másik megjelenése az ISA szinten a memóriasémák. Az csak természetes, hogy egy LOAD utasítás, amelyet egy STORE utasítás után hajtunk végre ugyanazon a címen, ugyanazt az értéket fogja erdményezni, amelyet az imént tároltunk. Ahogy a 4. fejezetben is láttuk, néhány megvalósításban a mikroutasítások át vannak rendezve. Így fennáll az a veszély, hogy a memória nem úgy fog viselkedni, ahogyan azt elvárjuk. A probléma csak súlyosbodik egy többprocesszoros rendszeren, ahol a CPU-k folyamatosan olvasni vagy írni akarnak a megosztott memóriába.
A rendszertervezők többféleképpen is megpróbálták megoldani ezt a problémát. Az egyik drasztikus megoldás, hogy a memóriaeléréseket sorba állítják, vagyis csak akkor jöhet a következő, mikor az előtte lévő művelet már befejeződött. Ez a módszer rontja a teljesítményt, de egyszerű megoldást jelent (minden művelet szigorú sorrendben hajtódik végre).
Ahhoz, hogy valami sorrend legyen a memóriában a programnak egy SYNC utasítást kell végrehajtania, amely letiltja új memóriahivatkozások fogadását egészen addig, míg az előző műveletek be nem fejeződtek. Ez súlyos terhet jelent a programozóknak, mert meg kell érteniük, hogyan is műküdik annak a gépnek a mikroarchitektúrája, de ez teljes szabadságot jelent a hardvertervezők számára a memóriaoptimalizálás terén.
Közbülső memóriamodellek is elképzelhetőek, ahol a hardver automatikusan blokkol bizonyos műveleteket. A mikroarchitektúra ezen sajátosságai rendkívül bosszantók (legalábbis az alacsony szinteken programozók számára), mindezek ellenére a fejlődés mégis ebbe az irányba halad a legújabb technológiák miatt (mikroutasítások átrendezése, pipeline,több cacheszint, stb). Még több ilyen rendellenes példát fogunk látni a fejezet hátralévő részeiben.

5.1.3 Regiszterek

Minden számítógépnek van néhány, az ISA szinten is látható regisztere. Ezeknek az a feladata, hogy szabályozzák a programok futását, ideiglenesen eredményeket tároljanak stb.. Általában azok a regiszterek, amelyek a mikroarchitektúra szintjén láthatóak (pl.: TOS és MAR, lásd 4-1-es ábra) nem láthatók az ISA szinten, bár némelyik mint például az utasításszámláló vagy a veremmutató regiszter mintkét szinten jelen van. Másrészről, az ISA szinten lévő regiszterek mindegyike látható a mikroarchitektúra szintjén is.
Az ISA regiszterek két csoportba sorolhatók: speciális és általános célú regiszterek. A speciális regiszterek tartalmazzák az utasításszámláló, veremmutató regisztereken kívül a többi olyan regisztert is, melyeknek valamilyen speciális funkciója van.

Ellentétben, az általános célú regiszterek azért vannak, hogy fontos változókat és számítások részeredményeit tárolják. Fő szerepük a gyakran használt adatok gyors elérésének biztosítása (alapvetően a memória kikerülésével). Egy gyors CPU-val és (viszonylag) lassú memóriával rendelkező RISC gépnek legalább 32 általános célú regisztere van, az újabb és újabb processzorokban pedig egyre több lesz.
A legtöbb számítógépben az általános célú regiszterek teljesen megegyeznek. Tehát ha a regiszterek mindegyike egyenértékű, akkor a fordító használhatja R1-et egy ideiglenes adat tárolására, de ugyanúgy használhatja az R25-öt is. Nem számít, melyiket választja.
Néhány gépnek vannak olyan általános célú regiszterei, amelyek egy kicsit mégis speciálisak. Például a Pentium II-n van egy EDX nevezetű regiszter, amely általános célokra is használható, de ez tartalmazza a szorzás végeredményének illetve osztásnál az osztandó szám "felét".

***[310-313]

Habár az általános célú regiszterek teljes mértékben felcserélhetők, mégis általános dolog az operációs rendszer vagy a fordítók számára, hogy elfogadjanak bizonyos egyezményeket arról, hogy ezen regisztereket hogyan használják. Például egyes regiszterek tartalmazhatnak paramétereket, amelyek egy meghívott eljáráshoz szükségesek, mások alkalmi regiszternek használhatók. Ha egy fordító egy fontos helyi változót tesz R1 – be majd meghív egy könyvtári eljárást, ami azt hiszi, hogy R1 egy alkalmi regiszter, amely számára elérhető, akkor mire a könyvtári eljárás visszatér R1 – ben valószínűleg szemét lesz. Ha vannak az adott rendszerre jellemző megállapodások a regiszterek használatáról, a fordítóknak és a gépi kódú programozóknak tanácsos ezeket követni.

Az ISA – szintű, felhasználói programok számára is látható regiszterek mellett mindig léteznek tekintélyes mennyiségű különleges célra fenntartott regiszterek is, amelyek csak kernel – módban elérhetők. Ezek a regiszterek vezérlik a különféle cache – ket, a memóriát, a be – és a kimeneti eszközöket és a gép más hardware jellemzőit. Ezeket csak az operációs – rendszer használja, így a fordítók és a felhasználók nem szükséges, hogy ismerjék őket.

Az egyik vezérlő regiszter, ami a kernel és a felhasználói regiszterek átmenete: a **jelző regiszter** vagy **PSW (Program Állapot jelző Szó)**. Ez a regiszter különféle, a cpu számára fontos biteket tartalmaz. A legfontosabb bitek az állapotkódok. Ezek minden ALU ciklus során beállítódnak, és a legutóbbi művelet helyzetét tükrözik. Jellemző állapotjelző bitek például:

N – értéke 1, ha az eredmény Negatív
Z – értéke 1, ha az eredmény nulla (Zéró)
V – értéke 1, ha az eredmény (Verem-)túlcsordulást okozott
C – értéke 1, ha végeredmény a bal szélső bit kivitelét okozta (Carry out)
A – értéke 1, ha a 3. bitet vitték ki (Auxilary carry = segéd kivitel)
P – értéke 1, ha a végeredménynek is van Peritása

Az állapotjelzők fontosak, mert az összehasonlító és feltételes elágazó utasítások (úm. feltételes ugró utasítások) használják. Például a CMP utasítás jellemzően kivonja egymásból a két paramétert, és a különbség alapján állítja be az állapotjelzőket. Ha a paraméterek egyenlőek, a különbség nulla lesz és a Z állapotjelző a PSW – ben 1 – re lesz beállítva. Egy ezt követő BEQ (elágazó egyenlőség) utasítás megvizsgálja a Z bitet, és ha az értéke 1, akkor elágazik.

A PSW az állapotjelző kódok mellett mást is tartalmaz, de a teljes tartalom gépről – gépre változik Jellemző kiegészítő mezők a gép – mód (pl. felhasználó/kernel), a nyomkövető bit (debugging során használt), a CPU elsőbbségi szint és a megszakítást engedélyező helyzet. A PSW általában olvasható felhasználó – módban, de néhány mező csak kernel – módban írható (pl.: felhasználó/kernel módjelző bit).

5.1.4 Utasítások

Az ISA szint tulajdonképpen gépi utasítások összessége. Ezek irányítják a gép működését. A memória és a regiszterek közötti adatmozgatás mindig LOAD és STORE utasításokkal történik, a MOVE paranccsal pedig egyik regiszterből a

másikba másolja az adatot. Számolási műveletek mindig jelen vannak, úgy mint igaz hamis utasítások vagy összehasonlító utasítások ahol az eredménytől függően a program elágazik. Már láttunk néhány tipikus ISA utasítást és ebben a fejezetben még többet meg fogunk ismerni.

5.1.5 Hogyan néz ki a PII ISA szintje

Ebben a fejezetben 3 teljesen eltérő ISA szintet fogunk megnézni: az Intel IA-32, ami a Pentium II – ben testesedik meg, V9SPARC ami az UltraSPARC processorban van, és a JVM a picoJava II –ben. Nem az a célunk, hogy kimerítő leírást készítsünk az ISA szintről, hanem kiemeljük a legfontosabb tulajdonságait, és megmutassuk, hogyan is néznek ki ezek a különböző változatokban. Kezdjük mindjárt a PII –vel.

PII processzort több generáció óta fejlesztik, elődeinek tekinthetők azok a legelsőprocesszorok is, amiről a legelső fejezetben beszéltünk. Az alap ISA nemcsak a 8086 és 8088 – as processzorokra írt programok futtatását támogatja, hanem tartalmaz maradványokat 8080 – as processzorból is ami egy a 70 – es évekbe népszerű cpu volt. A 8080 egy olyan processzor volt, aminek kompatibilisnek kellet lennie a 8008 – cal ami egy 4004 – en alapult ami egy 4 bites őskövület volt.

A szoftver szemszögéből a 8086 és a 8088 - as processzorok teljesen 16 bites gépek voltak. Habár 8088 – nak 8 bites adatbusza. Ezeknek az utóda a 8286 is 16 bites volt. A fő előnye a nagyobb címezhető területében rejlett, bár kevés program használta ezt, mert 16384 db 64k – s szegmensből állt nem pedig egy lineárisan címezhető 2^{24}bytos memória.

8386 – os volt az első 32 bites processzor az intelek közül. Minden ezt követő géptípus(8486,Pentium,Pentium Pro, PentiumII, Celeron, és Xeon) teljesen ugyanazzal a 32 bites architektúrával rendelkezik, mint a 8386, ezt hívjuk **IA – 32** – nek, és itt ezzel fogunk részletesen foglalkozni. A leglényegesebb változtatás a 8386 óta az MMX utasítások megjelenése volt a Peniumban és az azt követő processzorokban.

A PentiumII – nek háromféle működési módja van, kettő ebből arra szolgál, hogy kompatíbilis legyen a 8088 – cal. **Valós módban** az összes 8088 óta bevezettet tulajdonság ki van kapcsolva, így a Pentium II úgy viselkedik, mint egy mezei 8088, ha valamelyik programban valami hiba keletkezik, akkor az egész gép összeomlik. Ha az Intel embereket gyártana tenne beléjük egy olyan kapcsolót, amivel átváltoztathatja őket csimpánzzá (agy nagy része kikapcsol, nem beszél, legtöbbször banánr eszik stb.).

Egy lépéssel feljebb a 8086 – os **Virtuális mód** van, amely lehetővé teszi a 8088 –as programok biztonságosabb futtatását. Ebben a módban az operációs rendszer felügyeli az egész gépet. Egy régi 8088 – as program futtatásához az operációs rendszer létrehoz egy speciálisan elszigetelt környezetet, ami úgy viselkedik mint a 8088 kivéve azt az esetet ha a program összeomlik, ekkor az operációs rendszer jelez a helyett, hogy a rendszer összeomlana. Amikor a felhasználó a windowsban egy msdos ablakot nyit és ebben egy programot indít, akkor ez a program ebben a virtuális 8086 – os módban fog futni azért, hogy a windwos megvédhesse magát a rosszalkodó msdos programoktól.

Az utolsó mód a **Védett mód,** amelyben a Pentium II ténylegesen Pentium II ként viselkedik, nem pedig egy méregdrága 8088 – as ként. Ezen bellül is 4 szint (privilege levels) érhető el , amelyet a PSW bitjei szabályoznak. A 0. szint megfelel

egy másmilyen gép kernel módjának és a gép minden funkciója elérhető. Ezt az operációs rendszer használja. A 3. Szint a felhasználói programok számára van. Ez nem enged olyan utasításokat, amelyek biztos hibát okoznak, és a felügyelő regisztereket és így nem hagyja hogy a rosszul működő programok tönkretegyék a teljes rendszert. Az 1 – es és a 2 – es szintet ritkán használják.

A Pentium II – nek hatalmas címezhető területe van memóriája 16384 szegmensre van osztva és ezek mindegyike 0 tól a 2^{32}-1 - ig címezhető. A legtöbb operációs rendszer (beleértve a UNIXot és a WINDOWS összes verzióját) csak egy szegmenset támogat így a legtöbb alkalmazás is csak 2^{32} byte méretű lineáris memória területet lát. Ennek egy részét maga az operációs rendszer foglalja el. A címezhető terület minden bytja rendelkezik saját címmel, ami egy 32 bit hosszú szó. Szavak az ún. little endian formátumban vannak tárolva (azaz az alsó bytenak van a legkisebb címe).

A Pentium II regisztereit az 5-3 as ábra mutatja. Az első négy regiszter az EAX, EBX, ECX, EDX 32 bites és többé - kevésbé általános célú, habár mindegyiknek megvan a speciális feladata. EAX a fő számolási regiszter ; az EBX használható mutatók tárolására (memória címek) ; ECX fontos szerepet játszik a ciklusokban ; EDX pedig a szorzásban és az osztásban van szerepe, ahol az EAX – szel együtt tartalmazzák egy 64 bites számot, ami a szorzás eredményét illetve osztásnál az osztandót tartalmazza. Ezen regiszterek mindegyike tertalmaz egy 16 bites regisztert, ami a 32 bit alsó 16 bitje, és egy 8 bites regosztert ami a 32 bit alsó 8 bitje. Ezek a regiszeterek könnyűvé teszik a 16 és 8 bites mennyiségekkel való számolást. A 8088 és a 8286 csak a 8 és a 16 bites regisztereket ismeri. A 32 bites regiszterek 8386 ban jelentek meg elnvezésük elé egy E betű került, ami az angol extended (kiterjesztett) szóból származik.

A következő három regiszter is némileg általános célú, de ezek már sokkal inkább speciálisak mint az előző négy volt. Az ESI és az EDI regiszterek mutatókat tárolnak, például a hardveres tömbkezelő utasításokban, ahol ESI a forrás tömb EDI pedig a cél tömb címét tartalmazza. Az EBP szintén mutató, ún. bázismutató regiszter, olyan mint a LV az IJVM – ben. Végül az ESP a veremmutató.

5-3-as ábra: a PII elsődleges regiszterei

A regiszterek következő csoportja (CS – GS – ig) a szegmensregiszterek. Durván fogalmazva ezek elektronikus trilobiták, régi maradványok amelyek a 8088 memória címzésében játszottak szerepet. Elég annyit tudnunk, hogy amikor a PII egyszerű lineáris 32 bites címterületet használ, ezeket mellőzi. Következő az EIP, ami az utasításszámláló (Extended Instruction Pointer). Végül elérkeztünk az EFLAGS - hoz ami tulajdonképpen a PSW.

5.1.6 Hogyan néz ki az UltraSparcII ISA szintje?

A Sun Microsystems először 1987 – ben mutatta be a SPARC architektúrát. Elsők között volt a kereskedelemben megjelenő RISC processzorok között. Az architektura a 80 – as években a Berkeley Egyetem kutatásain alapszik (Patterson, 1985; Petterson és Séquin, 1982). Az eredeti SPARC egy 32 bites architektúra volt de az UltraSPARC II már 64 bites processzor, ami egy V9 – es SPARC – on alapul, ezt

fogjuk ebben a fejezetben tárgyalni.

***[314-317]
Nem érkezett meg. Szabó Péter: h838452

***[318-321]

A JVM-nek nincsenek olyan általános célú regiszterei, melyeket egy program betölthet, vagy tárolhat. Ez egy igazi virtuális gép. Bár napjainkban ez a megoldás szokatlannak számít, ez járul hozzá az egyszerű és elegáns ISA-hoz, és a könnyű összeállíthatósághoz.

A fő probléma a virtuális gépekkel, hogy nagyon sok memóriahivatkozás szükséges hozájuk. Azomban, ahogy azt már láttuk a 4.6.3-as szakaszban is, egy okos terv nagyrészben megsemmisül úgy, hogy összetett JVM utasításokat fűznek egybe. Rádaásul a felépítés egyszerűsége azt jelenti, hogy csekély mennyiségű szilíciummal megvalósítható, meghagyva a csiprész nagyrészének elérhetőségét a hatalmas első szintű cache memória részére, ami tovább csökkenti a memóriahivatkozások számát. (A pico-Java II-nek legfeljebb 16KB+16KB cache memóriája van, mert nem egy káprásztatóan gyors, hanem inkább egy nagyon olcsó csip létrehozása volt a cél.)

5.2 ADATTÍPUSOK

Minden számítógépnek szüksége van adatokra. Valójában sok számítógéprendszernek a valódi célja a pénzügyi, kereskedelmi, tudományos, műszaki és egyéb adatok feldolgozása. Ezeket az adatokat tudományos alakkal kell kifejezni a számítógépben. Az ISA szintjén változatos adattípusok használatosak. Ezeket az adattípusokat fogjuk bemutatni.

Egy bizonyos adattípusnál az egyik kulcskérdés mindig a hardveres támogatottság. A hardveres támogatottság azt jelenti, hogy létezik egy vagy több utasítás, amelyek egy bizonyos formájú adatra számítanak, és a felhasználó nem választhat szabadon egy másik adatformát. Például a könyvelőknek van egy olyan sajátságos szokásuk, hogy a negatív számoknál a mínusz jelet inkább a szám jobb oldalára írják, minthogy a bal oldalára, ahogy azt a számítógépszakemberek is tették. Gondoljunk arra, ha egy könyvelőcégnél megyőznék a számítógépközpont vezetőjét, hogy jobboldali mínuszjelet használjanak (a baloldali helyett). Ez kétségtelenül mély benyomást tenne a vezetőre - mivel az összes szoftver működésképtelenné válna (egy ilyen változtatás következtében). A hardver egy bizonyos formát vár az egészekre, és nem működik helyesen, ha valami mást kap.

Most vegyünk egy másik könyvelőcéget, amely éppen most kötött szerződést, hogy ellenőrzi az állami költségvetést (mennyivel tartozik az amerikai kormány az embereknek). Itt nem működne a 32 bites számolás, mert a számok nagysága nagyobb, mint 2^{32}-en (körülbelül 4 billió). Az egyik megoldás, hogy két 32 bites egészet használunk, így 64 bites lesz. Ha a gép nem támogatja az ilyenfajta **duplapontos** számokat, akkor szoftverrel kell megoldani minden számolást, de a két mód bármelyike jó, mert a hardvernek mindegy. Ez egy olyan adattípusra példa, melyet a hardver nem támogat, de nem igényel hardvermódosítást. A következő fejezetekben olyan adattípusokat fogunk vizsgálni, melyeket a hardver támogat, és hogy melyik milyen speciális formát igényel.

5.2.1 Numerikus adattípusok

Az adattípusok két kategóriára oszthatók fel: numerikus és nem numerikus. A legfőbb numerikus adattípus az egészek (integers). Sokfajta hosszúságban léteznek, tipikusan 8, 16, 32 és 64 bitesek. Az egészek dolgokat számolnak (pl: egy vasárúbolt raktárán lévő csavarhúzók száma), azonosítanak (pl: banki számla száma) és sokminden másra alkalmasak. A legtöbb modern számítógép az egészket "páros" (páros: negatív számok is) bnális jelrendszerben tárolja, bár a múltban más rendszerek is használták. A bináris számok tárgyalása az A Függelékben található.

Néhány számítógép támogat jelöletlen egészeket hasonlóan a jelöltekhez. A bitek nincsenek előre megjelőlve (helyiérték), és minden bit tartalmaz adatokat. Ennek az adattípusnak az az előnye, hogy így van egy plussz bit, így például egy 32 bites változó 0-tól (2^{32}-1)-ig bezárólag tud tárolni egészeket. Ellentétben a "páros" jelölt 32 bites egésszel, amely csak (2^{31}-1)-ig tudja kezelni a számokat, de természetesen negatív számok kezelésére is alkalmas.

Vannak olyan számok, amelyek nem fejezhetők ki egészekkel, ilyen a 3,5. Az ilyen számok kifejezésére használjuk a **lebegőpontos** számokat. Ezeket a számokat a B Függelékben tárgyaljuk. Ezek hossza 32, 64, olykor 128 bit. A legtöbb számítógépnek vannak utasításai a lebegőpontos számolásra. Sok számítógépnek külön regiszterei vannak az egész változók és a lebegőpontos változók tárolására.

Néhány programnyelv, nevezetesen a COBOL, decimális számokat is megenged adattípusként. Azok a gépek, amelyek COBOL-barátok akarnak lenni, gyakran hardveresen támogatják a decimális számokat. Ez speciállisan úgy történik, hogy a tizes számrendszerbeli számjegyeket 4 bittel kódólják, és két számjegy kerül egy bájtba (bináris kód, decimális forma). Azonban a számolás nem működik helyesen a kreált decimális számokon, ezért speciállis decimális-számolás-javító utasításokra van szükség. Ezeknek az utasításoknak tudniuk kell az átvitelt a 3. bitből. Ez az, amiért a feltételkód gyakran tartalmaz egy kiegészítő átviteli bitet. Mellékesen a rossz hírű 2000. év problémáját azok a COBOL programozok idézték elő, akik úgy gondolták, olcsóbb az évet 2 decimális számjeggyel kifejezni, mint egy 16 bites bináris számmal. Egy kevés optimalizálás.

5.2.2 Nem numerikus adattípusok

Bár a legtöbb korai gép a számokkal való műveletekkel kereste kenyerét, a modern számítógépeket gyakran használnak nem numerikus alkalmazásokra, mint a szövegszerkesztés vagy az adatbáziskezelés. Ezekhez az alkalmazásokhoz más adattípusokra és gyakran más ISA-szintű utasítások támogatására van szükség. A karakterek nyílvánvalóan fontosak, bár nem minden számítógép gongoskodik hardveres támogatásukról. A legnépszerűbb karakterkódrendszerek az ASCII és az UNICODE. Ezek 7 illetve 16 bitesek. A 2. fejezetben tárgyaltuk ezeket.

Nem szokatlan, hogy az ISA-szintjén olyan utasítások legyenek, melyek a sztringek kezelésére lettek tervezve. Sztring: egymás után következő keraktersorozat.

Az ilyen sztringeknek sokszor sokszor egy speciállis karakter szab határt, mely a sztring végén áll. Egy sztring hossza magába foglalhatja a végjelet is. Az utasítások képesek másolni, keresni, szerkeszteni és egyéb funkciókat végrehajtani a sztringeken.

A logikai értékek szintén fontosak. A logikai érték kétféle értéket vehet fel: igaz vagy hamis. Elméletben 1 bit elég lenne kifejezni a logikai értéket, például 0-val a hamisat és 1-gyel az igazat (vagy fordítva). A gyakorlatban 1 bájtot használnak egy logikai érték kifejezésére, mert az egyedüli biteknek nincs saját címzésük egy bájtban és így nehéz kódolni őket. Közös megegyezés szerint a 0 hamisat, és minden más az igazat jelent.

Az egyik helyzet, amikor a logikai értéket természetszerűen 1 bittel fejezzük ki, amikor egy egész sor van belőle, tehát 32 logikai értéket 32 biten tárolhatunk. Az ilyen adatszerkezetet **bit térkép**nek hívjuk, és sokféle összetételben előfordul. A bit térkép például egy lemezen lévő üres szektorok megjelölésére szolgálhat. Ha egy lemezen n üres szektor van, akkor a bit térkép n bites.

Az utolsó adattípusunk a pointer, ami csak egy gépi címzés. Már láttunk ismétlődő pointereket. A Mic-x gépen az SP, PC, LV és a CPP példák a pointer adattípusra. Egy rögzített határon belül változtatható pointerek kódolása rendkívül népszerű minden gépen.

5.2.3 A Pentium II adattípusai

A Pentium II támogatja a "páros" jelölt egészeket, a jelöletlen egészeket, a bináli kódolású decimális számokat és az IEEE 754 lebegőpontos számokat, ahogy az fel van sorolva az 5-6.-os ábrán. Eredete miatt, minthogy egy 8/16 bites gép, jól kezeli az ilyen nagyságú egészeket, a számos utasítással együtt. Ezek az utasítások számolásra, logikai műveletek elvégzésére és összehasonlításra használatosak. A változókat nem kell felsorolni a memóriában, de jobb megvalósítást érünk el, ha a változók címzései 4 bájt többszörösei.

Típus	8 bit	16 bit	32 bit	64 bit	128 bit
Jelölt egész	x	x	x		
Jelöletlen egész	x	x	x		
Binális kódolású decimális	x				
Lebegőpontos			x	x	

Ábra 5-6. A Pentium II numerikus adattípusai. A támogatott adattípusokat x-szel jelöltük.

A Pentium II a 8 bites ASCII karakterek kezelésében is jó. Vannak speciállis utasításai másolásra és keresésre a sztringekben. Ezeket az utasításokat olyan sztringekre lehet használni, melyeknek ismerjük a hosszát, vagy melyeknek a vége meg van jelölve. Gyakran használják ezeket a sztringeket kezelési könyvtárakban.

5.2.4 Az UltraSPARC II adattípusai

Az UltraSPARC II széles körben támogatja az adattípusokat, ahogy azt az 5-7.-es ábra is mutatja. Az egészekhez kizárólag 8, 16, 32 és 64 bites változókat támogatja mind jelölt, mind jelöletlen típusban. A jelölt egészek “páros”-ak. A 32, 64 bites változókat tartalmazza és megerősíti az IEEE 754 szabvány (32 és 64 bites számokhoz). A binális kódolású decimális számokat nem támogatja. Minden változót fel kell sorolni a memóriában.

Típus	8 bit	16 bit	32 bit	64 bit	128 bit
Jelölt egész	x	x	x	x	
Jelöletlen egész	x	x	x	x	
Binális kódolású decimális					
Lebegőpontos			x	x	x

Ábra 5-6. Az UltraSPARC II numerikus adattípusai.

Az UltraSPARC II erősen regiszterorientált, és majdnem minden utasítás 64 bites regisztereken működik. Karakter és sztring adattípusokat nem támogatnak a speciális hardver utasítások.

5.2.5 A Java Virtialis Gép (JVM) adattípusai

A Java erősen tipizált nyelv, ami azt jelenti, hogy minden változónak speciális típusa és mérete van, amit ismerünk a deklarációkor. A JVM által támogatott runtime típusok ezeket a típusokat tükrözik vissza. A JVM az 5-8.-as ábrán felsorolt típusokat támogatja. A jelölt egészek “páros”-ak. Jelöletlen egészeket nem nyújt a Java nyelv és nem is támogatja. Valamint a binális kódolású decimális számokat sem.

Típus	8 bit	16 bit	32 bit	64 bit	128 bit
Jelölt egész	x	x	x	x	
Jelöletlen egész					
Binális kódolású decimális					
Lebegőpontos			x	x	

Ábra 5-6. A JVM numerikus adattípusai.

A JVM támogatja a karaktereket, de a hagyományosabb 8 bites ASCII karakterek helyett a 16 bites UNICODE karaktereket. Ez korlátozott támogatás a pointereknek, ami föleg belső használatú deklarációra alkalmas, de a runtime rendszer felhasználói programként nem tud közvetlenül pointereket kódolni. A pointereket a gép főleg a feladatok besorolására használja.

***[322-325]

Bugyi Zsolt - prog. mat. I

5.3 Utasítás formák

Az utasítás az opcode-ból áll, rendszerint néhány egyéb információval együtt, mint például honnan érkeznek az operandusok és hova érkezik az eredmény. Az operandusok helyének (azaz a címük) meghatározásának fő témáját címzésnek nevezzük, és most ezt fejtjük ki.
Az 5-9. ábra néhány lehetséges 2-es szintű utsítás mutat be. Az utasításoknak mindig van egy opcode-ja, hogy leírja mit csinál az utasítás. Szerepelhet nulla, egy, kettő vagy három cím.

5-9. ábra Négy gyakori utasításformátum: (a) utasítás cím nélkül. (b) utasítás egy címmel (c) utasítás két címmel (d) utasítás három címmel

Némely gépen, minden utasítás egyforma hosszú. Más gépeken sok más hossz lehetséges. Lehetnek szó hossznál rövidebb, azonos vagy hosszabb utasítások. Az utasítások azonos hossza egyszerűsíti és könnyíti a dekódolást, de gyakran pazarolja a helyet, mivel minden utasítás a szükséges hossznál hosszabb leghosszabb utasítás hosszát veszi fel. Az 5-10-es ábra néhány lehetséges összefüggést mutat az utasítás és a szó hossza között.

5-10. ábra Néhány lehetséges összefüggés az utasítás és a szó hossz között

5.3.1 Az utasítás formátumának tervezési kritériuma
Ha egy számítógéptervező csapatnak utasításformákat kell választania a géphez tekintettel kell lennie a tényezők számára. Nem szabad alábecsülni a döntés nehézségét. Az utsításformáról való döntést még az új számítógép tervezése elején meg kell hozni. Ha a számítógép a kereskedelemben sikeres, akkor az utasításkészlet

20 vagy akár több évig is fennmaradhat. Fontos az új utasításokkal való bővítés kihasználása és más lehetőségek kihasználása, melyek felmerülnek egy hosszabb időszak alatt, de csak akkor sikeres az architektúra, ha hosszabb időn át marad fenn.
Egy különleges ISA hatékonysága nagymértékben függ a technológiától melyjel a számítógépet kivitelezik. Hosszabb idő alatt óriási változáson fog átmenni és néhány ISA választás nem fog szerencsésnek tűnni. Például, ha a memóriaelérés gyors, akkor a stack alapú tervezés (mint a JVM) jó választás, de ha lassú, akkor sok regiszternek (mint az Ultra SPARCII) mennie kell. Azoknak az olvasóknak, akik úgy gondolják, hogy könnyű a választás, azok fogjanak egy papírdarabot és írják le jóslataikat 20 évre előre: (1) egy jellmező CPU órasebességet, és (2) egy jellemző RAM elérési időt a számítógépekre. Hajtsák össze a cetlit és tartsák meg 20 évig. Majd hajtsák szét és olvassák el az idő leteltével.
Persze, még az előrelátó tervezők sem hozhatnak mindig helyes döntést. És ha mégis, akkor is szembe kell nézniük a rövid határidőkkel is. Még ha ez az elegáns ISA még csak egy kicsivel drágább is, mint ronda versenytársai, akkor sem sikerülhet túl sokáig fenntartani az elegáns ISA megbecsülését.
Ha minden megegyezik, akkor a rövid utasítások jobbak mint a hosszabbak. Ha egy program n 16 bites utasításból áll, akkor csak fele akkora memóriát foglal el mint n 32 bites utasítás. Az egyre csökkenő memória árak miatt ez a tényező kevésbé lehet fontos a jövőben, eltekintve attól, hogy a szoftver helyigénye még a memória árának esésénél is gyorsabban növekszik.
Azonkívül, az utasítások méretének minimalizálásával a dekódolás és az átlapolás nehezebbé válik. Ezért a minimális utasításméret megvalósítását ellensúlyozni kell azzal az idővel mely az utasítások dekódolásához és futtatásához szükséges.
Más okból az utasításhossz minimalizálása máris fontos és a jövőben még fontosabb lesz a gyorsabb processzorok esetében: memória sávszélessége (a memória által szolgált bit/sec száma). A processzorok sebességének jelentős növekedése az utóbbi évtizedben nem egyezett meg a memória sávszélességének növekedésével. Egyike az egyre inkább gyakori problémáknak a processzorokkal kapcsolatban abból ered, hogy a memóriarendszer képtelen az utasításokat és változókat olyan gyorsan szolgáltatni, mint ahogy a processzor képes lenne felhasználni azokat. Minden memóriának van egy sávszélessége, melyet a technológiája és mérnöki tervezése határozott meg. A sávszélesség problémája nemcsak a főmemóriára vonatkozik, hanem a cache-re is.
Ha a sávszélessége egy utasítás-cache-nek t bps és az átlagos utasításhossz r bit, akkor a cache legfeljebb t/r utasítást tud másodpercenként küldeni. Megjegyzendő, hogy ez egy felső határa a processzor sebességének amellyel utasításokat futthathat, habár jelenleg vannak kutatási munkák, melyeknek célja, hogy áttörjék még ezt az akadályt is. Nyilvánvaló, hogy a sebességet, melyen az utasítások lefuthatnak (mint például a processzor sebessége) korlátozhatja az utasítás hossza. Rövidebb utasítás gyorsabb processzort jelent. Amióta a modern processzorok alkalmasak több utasítás végrehajtására órajelenként, ezért órajelenként több utasítást kell továbbítani. Ebből a szemszögből nézve az utasítás-cache-t az fontos tervezési kritériummá teszi az utasítások méretét.
A második tervezési kritérium elegendő hely biztosítása az utasításformában minden kért művelet kifejezéséhez. Olyan gép mely 2^n műveletet ismer és minden utasítása kisebb mint n bit nem létezik. Egyszerűen nem lesz elég hely az opcode-ban jelezni, hogy melyik utasítás szükséges. És a múlt is újra és újra mutatja azt a könnyelműséget, hogy nem hagynak szabadon lényeges mennyiségű opcode-ot a jövőbeni utasításkészlet bővítéséhez.

A harmadik kritérium a cím mezőben lévő bitek számára vonatkozik. Tekintsük a tervezését egy gépnek, mely 8 bites jellegű, és egy memóriának ami 2^32-en karaktert kell tartalmaznia. A tervezők választhatnak az egymást követő címek egységekhez való rendelésénél 8, 16, 24 vagy 32 bites, valamint más lehetőség közül.
Képzeljük el mi történhet, ha a tervező csapat két szemben álló pártra oszlik, az egyik azt javsolja, hogy a 8 bites byte legyen a memória alapegysége, míg a másik a 32 bites szót javasolja a memória alapegységének. Az előbbi csapat 2^32 byte memóriát feltételez, számozottan 0, 1, 2, ..., 4,294,967,295. Az utóbbi csapat 2^{30} szónyi memóriát feltételez, számozottan 0, 1, 2, ..., 1,073,741,823.
Az első csapat arra mutatna rá, hogy két karakter összehasonlításához a 32 bites szervezésben a programnak nem csak a karaktert tartalmazó szót kell előhívnia, hanem ki kell hogy bontsa mindegyik karaktert az őt tartalmazó szóból, hogy összehasonlíthassa őket. így ez több utasításba kerülne, és helyet pazarolna. A 8 bites szervezés más oldalról minden karakter számára gondoskodik címről, így sokkal egyszerűbb az összehasonlítás.
A 32 bites szó támogatói azzal vágnának vissza, hogy rámutatnak, hogy az ő javaslatuknál csak 2^{30}-adikon elkülönített címre van szükség, a cím megadásának hossza csak 30 bit, ahol ahogy a 8 bites javaslatnál 32 bit-re van szükség ugyanazon memória címzéséhez. Rövidebb cím rövidebb utasítást jelent, ami nem csak kevesebb helyet foglal, de kevesebb időre is van szükség az adatlekéréshez. Választhatóan, megtarthatják a 32 bites címzést hivatkozva a 16 GB-os memóriára a kicsi 4 GB memória helyett.
Ez a példa azt mutatja, azért hogy jobb memóriafelbontást nyerjünk, egyiknek fizetnie kell az árat a hosszabb címekért, és éppen ezért azok hosszabb utasításaikért. A legalapvetőbb felbontás az a memória szervezés, ahol minden memóriabit közvetlenül címezhető (pl. a Burroughs B1700). Másik szélsőséges eset az a memória, ami nagyon hosszú szavakból áll (pl. a CDC Cyber series 60 bites szavakkal).
A modern számítógépes rendszer arra a kompromisszumra jutott, hogy bizonyos értelemben mindkettőből átveszi a rosszat. Megkövetelik az összes bitet ami szükséges a sajátos byte-ok címzéséhez, de mindem memóriaelérés egy, kettő, vagy néha négy szót olvas be egyidőben. Egy byte olvasása a memóriából az UltraSPARC II-n, például minimálisan 16 byte-ot tölt be (lásd: 3-47 ábra) és valószínűleg egy 64 byte-os cache sor egészét.

5.3.2 Az opcode kibővítése

Az előző részben láthattuk, hogy hogyan állt szemben egymással a rövid címzés és a jó memóriafelhasználás. Ebben a részben vizsgáljuk meg ezeket az ellenhatásokat beleértve mind az opcode-okat és mind a címeket. Nézzünk egy (n+k) bites utasítást, ahol k bites az opcode és az egyetlen cím n bites. Ez az utasítás 2^k különböző utasítást és 2^n címezhető memóriacellát enged meg. Választhatóan ugyanaz az n+k bit felosztható (k-1) bites opcode-ra, és (n+1) bites címre, azaz fele annyi utasítás mellett kétszer annyi memória címezhető, vagy ugyanakkora memória címezhető kétszer akkora felbontás mellett. Egy (k+1) bites opcode és egy (n-1) bites cím több műveletet eredményez, de kevesebb címezhető cella árán, vagy szegényebb felbontás és ugyanazon címezhető memória mellett. Elég bonyolult egyeztetések lehetségesek opcode bit és cím bit közt éppúgy, mint a fent leírt egyszerűbbeknél. A következő bekezdésekben tárgyalt változatot opcode bővítésének nevezzük.
Az opcode bővítésének fogalmát leginkább példán keresztül szemléltethetjük. Vegyünk egy gépet melyben 16 bit hosszú utasítások és 4 bit hosszú címek vannak,

ahogy az 5-11. példa mutatja. Ez az eset indokolt lehet egy gépnél, melynek 16 regisztere van (ezért 4 bit regiszter cím) amin minden aritmetikai művelet helyet foglal. Az egyik tervezet lehetne egy 4 bites opcode és 3 cím minde utasításban, így lesz 16 háromcímes utasításunk.

5-11. ábra utasítás 4 bites opcode-dal és három 4 bites címmezővel

Azonban, ha a tervezőknek 15 háromcímes utasításra ,14 kétcímes utasításra, 31 egycímes utasításra és 16 cím nélküli utasításra van szükségük, akkor használhatnak olyan opcode-ot 0-tól 14-ig háromcímes utasításként, de az opcode 15-öt másképpen értelmezik (lásd: 5-12 ábra).

Opcode 15 azt jelenti, hogy az opcode a 8-15. biten foglal helyet a 12-15. bit helyett. 0-3. bitek és 4-7. bitek szokás szerint két címet alkotnak. A 14 kétcímes utasítások mindegyikének 1111 van a baloldali 4 bitjén és 0000-tól 1101-ig tartó számok lesznek kezelve speciálisan a 8-11 biteken. Úgy lesznek kezelve, mintha az opcode-juk a 4-15 biten lenne. Az eredmény 32 új opcode. Mivel csak 31 szükséges, ezért a 111111111111 opcode úgy van értelmezve, hogy a tényleges opcode a 0-15. biten van, ez 16 cím nélküli utasítást erdményez.

Ezeken a tárgyalásokon végighaladva az opcode egyre csak hosszabodott: a háromcímes utasításoknak 4 bites opcode-juk van, a kétcímes utasításoknak 8 bites opcode-juk van, az egycímes utasításoknak 12 bites opcode-juk, és a nulla címes utasításoknak 16 bites opcode-juk.

***[326-329]

Az utasításkód kiterjesztésének ötlete bemutatja, hogyan lehet cserélgetni az utasításkódot és más információkat. A gyakorlatba, az utasításkódok kiterjesztése nem olyan világos és szabályos mint a mi példánkban. Valójában a változtatható hosszúságú utasítások at kétféleképpen is ki lehet aknázni. Az első, hogy az utasítások mind azonos hosszúságuak és a legkevesebb bitet azok az utasítások kapják melyek amelyek legtöbb bitet igénylik más dolgok meghatározására. A második, hogy az átlagos utasításhosszt lehet minimalizálni úgy hogy a gyakrabban használt utasítások rövidebbek a kevésbé gyakran használtakhosszabbak.

A változó hosszúságú utasításkódok ötletének legszélsőségesebb esetében lehetséges az átlagos utasításhossz minimalizálása az összes utasítás kódolásával, hogy minimalizálják a szükséges bitek számát. Sajnos ez azt eredményezte, hogy a változó hosszúságú utasításokat nem lehet byte határra igazítani. Bár voltak szabványok amelyeknek megvolt ez a tulajdonsága (például balsorsú Intel 432), a gyors dekódolás fontossága annyira fontos, hogy az ilyenfokú optimalizálás nem eléggé hatékony. Mindazonáltal gyakran alkalmazzá byte szinten. Később majd megvizsgáljuk a JVM szabványt és láthatjuk milyen körültekintően választották meg az utasításokat, hogy minimalizálják a program méretét.

5.3.3 A Pentium II utasításainak szerkezete

A Pentium II utasításainak szerkezete nagyon összetett és szabálytalan, hat változó hosszúságú mezővel melyek közül 5 opcionális. Az általános mintát az 5- 13-as ábra mutatja. Ez a helyzet azért alakúlt ki mert az architektúrát generációkon keresztül fejlesztették és néhány korai rosz választást is tartalmazott. A lefelé való kömpatibilitás miatt ezeket a korai döntéseket nem lehetett később visszavonni. Álltalában a kétoperandusu műveleteknél, ha az egyik operandus a memóriában van akkor a másik nem lehet ott. Így léteznek olyan utasítások, melyek összeadnak két regisztert, egy meóriában lévő adatot regiszterrel, és fordítva, de olyan nincs amely két memóriában lévő szót adna össze.

A korai Intel architektúrákban minden utasításkód 1 byte hosszú volt, ámbár a prefix byte-ot széles körben használták néhány utasítás módosítására. A prefix byte egy extra utasításkód toldalék az utasítás elején mely megváltoztatja hatását. A IJVM és a JVM WIDE utasítása példa a prefix byte-ra. Sajnos a fejlesztések néhány pontján az Intel kifutott az utasításkódokból ezért egy kód, a 0xFF, kijelölték mint váltókódot, hogy így engedélyezzenek egy újjabb utasításbyte-ot.

A Pentium II utasításkódjaiban lévő bitek egyenként nem adnak újjab információt az utasításról. Az egyetlen struktúra az utasításkódmezőben a legalacsonyabb bit használata, melyet néhány utasításban a byte/word jelzésére használnak, és a közvetlen mellette levé bit amely azt mutatja, hogy a memóriacím (ha van) a forrás, vagy a cél cím. Így álltalában az utasításkódot teljesen vissza kell fejteni ahoz, hogy milyen osztályú utasítást kell végrehalytani és így ahoz is, hogy milyen hosszú az utasítás. Ez teszi nehézzé a nagyteljesítményű végrahajtást, mivel költséges dekódolásra van szükségmég ahoz is, hogy megtudjuk, hol kezdődik a következő utasítás.

A legtöbb utasításnál, melyek egy memóriában lévő operandusra hivatkoznak, az utasításkódot egy második byte követi ami mindent elmond az operandusról. Ez a nyolc bit fel van osztva egy 2 bites MOD mezőre éskét 3 bites mezőre, REG és R/M.

Néha e byte első 3 bitjét az utasításkód kiterjesztéseként használják így 11 bites utasításkódot elérve. A 2 bites MOD mező azt jelentihpgy csak négyféleképpen lehet megcímezni egy operandust és egyik operandusnak regiszternek kell lennie. Logikailag az EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP közül bármely meghtaározható mint akármelyik regiszter, de a kódolás megtílt néhány kombinációt és azokat speciális esetekhez használja. Néhány mód igényel egy további byte-ot, melyet SIB-nek neveznek (Scale,Index,Base), a további meghatározáshoz. Ez a séma nem ideális, de kompromisszumot jelent lefelé való kompatibilitás és az újabb nem előrelátott tulajdonságok megvalósítása között.

Mindezekhez még hozzátartozik, hogy néhány utasításnál 1,2 vagy 4 byte határoz meg egy memóriacímet (eltolás) és talán mégegyszer 1,2 vagy 4 byte tartalmaz egy konstanst (azonnali operandus).

5.3.4 Az UltraSPARC II utasításainak szerkezete

Az UltraSPARC II ISA szabványával továbra is ragaszkodik a 32 bites utasításokhoz. Az utasitások általában egyszerűek és csak egyetlen műveletet határoznak meg. Egy tipikus aritmetikai művelet két regisztert határoz meg a forrás operandusokhoz és egyet a célhoz. Egy változat lehetővé teszi , hogy az utasítás egy 13 bites előjeles konstanst használjon az egyik regiszter helyett.

A LOAD két regisztert (vagy egy regisztert és egy 13 bites konstanst) ad ossze, hogy meghatározza azt a memóriacímet amit ki kell olvasni. Az adatot pedig a harmadik regiszter által meghatározott címre írja.

Az eredeti SPARC-nak nagyon korlátozott számú utasításformátuma volt, melyeket az 5-14-es ábra mutat. Idővel újabb típusokat adtak hozzá. Amikor e sorokat írjuk számuk 31 és folyamatosan nő. (Nem lehetünk messze attól, amikor néhány cég azt fogja reklámozni, hogy "A világ legosszetetteb RISC gépe." A legtöbb új változatot úgy nyerték, hogy levagdostak néhány bitet a különböző mezőkből.

Például az eredeti elágazások a 3. formátumot használták 22 bites eltolással.

Amikor a megjövendölt elágazásokat bevezették akkor a 22 bitből hármat levágtak.

Egyet használnak a jövendölés eldöntésére (elfogadva/nem elfogadva), és kettőt arra használnak, hogy meghatározzák melyik feltételkód biteket kell használni.

Így 19 bit maradt az eltolásra.Másik példa: több utasítás is van az adattípusok közötti konvertálásra(egeszet lebegőpontosra stb.) Több ezek közül az 1b forma egy változatát használja, az IMMEDIATE mezőt osztották föl egy 5 bites mezőre amely a forrás regisztert adja és egy 8 bitesre amely további utasításkód biteket ad. Az utasítások többsége viszont továbbra is az ábrán látható formákat használja.

Minden utasítás első két bitje segít meghatározni az utasítás formátumát és megadja a hardware-nek hogy hol találja az utasításkód többi részét, ha van még.

Az 1a formánál mindkét forrás regiszter, az 1b formánál pedig az egyik forrás regiszter a másik pedig egy -4096 és +4095 közé eső konstans. A 13. bit választ köztük. (A legjobboldalibb bit 0.) Mindkét esetben a cél egy regiszter.

Elégséges hely áll rendelkezésre 64 utsítás kódolásához és ezek közül néhány jövöbeli használatra van fenntartva.

Csak 32 bites utasításokkal nem lehetnének 32 bites konstansok az utasításokban.

A SETHI utasítás 22 bitet állít be helyet hagyva egy kovetkező utasításnak ami a többi 10 bitet állítja be. Ez az egyetlen utasítás ami ezt a formát használja.

A nem előrejelzett feltételes elágazások a 3. formát használják, melyeknek a COND mező mondja meg, hogy melyik feltételeket kell tesztelni. Az A bitnek a delay slotok elkerülésében van szerepe néhány feltételnél. Az előrejelzett elágazások a ugyanezt a formátumot használják kivéve a 19 bites eltolást ahogy az fentebb említettük.

***[330-333]

Lakos Péter 330-333.

5.3.5 JVM utasítás formátumok

A legtöbb JVM utasítás formátum különösen egyszerű az 5.-15.-ös ábrán látható valamennyi.Egyszerűségük valószínüleg abból adódik, hogy a JVM új.De várjunk csak 10 évet. Valamennyi utasítás egy 1 bájtos opcode-dal kezdődik.Néhányukban ezután egy második bájt következik, vagy egy index(mint az iload), egy állandó (constans, mint a bipush) vagy egy adat típusú jelző (indicator) (mint a newarray, amely a jelzett típus 1 dimenziós elrendezését képezi a halmazból).

A 3-as formátum lényegében megegyezik a kettessel, azzal az eltéréssel, hogy ebben 16 bites állandó (konstans) szerepel a 8 bites helyett.(pl. WIDE ILOAD vagy GOTO).A 4-es formátum csak IINC-re használatos.

Az 5-ös formátum csak MULTINEWARRAY-ra használatos, amely több dimenziós elrendezést készít a halmazból.

A 6-os formátumot csak az INVOKEINTERFACE használja, amely eljárás(módszer) csak bizonyos körülmények között működik. A 7-es formátumot csak a WIDW IINC használja, amely gondoskodik egy 16 bites indexről és egy 16 bites constansról,ezeket különböző szelvényekhez teszi hozzá. A 8-as formátumot csak a WIDE GOTO és a WIDE JSR parancsok használják (tartalmazzák), melyek távoli elágazásoknak adnak helyet és bizonyos eljárásokat hívnak életre.Az utolsó formátumot csak két parancs használja,mindkettő egy Java állítás-választó eszközt használ.Röviden, 8 specifikus parancs kivételével minden JVM utasítás használja az 1-es , 2-es, vagy a 3-as formátumot, amelyek rövidek és egyszerűek.

A valóságban a helyzet még sokkal egyszerűbb.A Java Virtual Machine utasításokat specifikusan úgy kódolták, hogy a legtöbb általános parancsot egyetlen bájt tartalmazza.Lehetőség van ara , hogy 256 utasítás elférjen egyetlen bájtban, sok közülük egyetlen utasítás általános formájának speciális változata, amelyekkel az IJVM-ben találkozhattunk.

Példaként lássuk, hogyan adhatunk hozzá egy helyi változót a Java-hoz. Ezt 3 különböző módon tehetjük meg, közülük a legrövidebb az általában használatos forma , míg a leghosszabb lefedi az összes lehetséges esetet. A JVM-nek van egy utasítása, az ILOAD, amely egy 8 bites indexet használ, hogy a helyi változót hozzáadja a halmazhoz. Azt is bemutattuk, hogyan teszi lehetővé a WIDE azt, hogy ugyanaz az opcode használható legyen arra, hogy az első 65,536 címszó(bejegyzés) bármelyikét meghatározza (részletezze,felsorolja)a helyi változó keretében. Mindazonáltal a WIDE ILOAD használata 4 bájtot igényel, 1 a WIDE-hoz, 1 az ILOAD-hoz és 2 további a 16 bites indexekhez.Ezt a felosztást az a tény indokolja, hogy az ILOAD jórésze az első 256 helyi változó egyikét használja. A WIDE prefixum(elöljáró) általában szükséges, de csak ritkán használatos.

De a JVM még ennél is továbbmegy. Mivel az eljárás jellemzői a helyi változó keretének első néhány szavához tartoznak, az

ILOAD-ot legáltalánosabban az alsó indexel rendelkező belépésekhez alkalmazzuk.JVM tervezői elhatározták, hogy ezek a elhelyezkedések annyira általánosak lesznek, hogy érdemes meghatározni minden egyes kombináció számára a különböző, eltérő 1bájtos opcode-okat.Az ILOAD_0(0x1A) a 0-s helyi változót teszi hozzá a halmazhoz.Ez pontosan megegyezik a 2 bájtos ILOAD 0 paranccsal, azzal a különbséggel ,hogy 1bájtot igényel a 2 helyett. Hasonlóan az ILOAD_1,ILOAD-2 és

az ILOAD-3(opcode 0x1B,0x1C és 0x1D)az 1-es, 2-es,és a 3-es helyi változót adja hozzá a halmazhoz. Jegyezzük meg, hogy az 1-es helyi változó például három különböző módon adható hozzá: LOAD_1-gyel, ILOAD 1-gyel vagy WIDE ILOAD 1-gyel.
Számos utasításban fordulnak elő hasonló variációk. Egyes speciális utasítások pontosan egyenértékűek lehetnek a BIPUSH-sal az 1,2,3,4 és 5-ös értékek miatt, csakúgy , mint az -1-nél.Szintén speciális utasítások vannak a halmaz változóinak tárolásához a helyi változó készlet első 4 szavából.
Érdemes tudni, hogy ezek a lehetőségek nem használhatóak szabadon.Kizárólag 256 egyedülálló utasítás sorolható
(határozható meg) egyetlen bájtban. Az elérhető 256 parancsból négyet az első 4 hely feltöltésére használhatunk fel.Megjegyzendő, hogy íly módon az alap ILOAD parancsokhoz további 5 parancs vehető fel.Természetesen a WIDE prefixum szintén felvehet egyet az elérhető 256 érték közül(és ez még nem utasítás, csak egy prefixum), de sokkal általánosabb, hogy számos opcode-ot is használ.
A JVM tervezői némileg eltérő eljárást alakalmaztak ahhoz, hogy meghatározzák a konstans készletből az operanduszok betöltését. Egyetlen utasítás két formájáról is gondoskodtak: az LDC-ről és az LDC-W-ről. A második forma, amelyet az IJVM is tartalmaz, az általános forma. A 65,536 szó közül bármely elérésére képes, amelyek a konstans készletben megtalálhatóak. Az első forma csak egy egyetlen bájtos indexet tartalmaz, ez csak az első 256 szó bármelyikét képes elérni. Ezen első 256 szó elérése történhet egy 2 bájtos utasítással, míg egyéb szavak csak 3bájtos parancsokkal érhetőek el.Ez a két választási lehetőség kettőt igényel a 256 opcode közül. Ha a tervezők ugyanazt a technikát használták volna, amelyet az ILOAD esetében is, nevezetesen, hogy inkább a WIDE prefixumot használják az LDC-W parancs helyett, a parancsok jóval érthetőbbek és szabályosabbak lennének. Mindez azt jelenti, hogy egyéb konstansok elérése az első 256-on kívül 4 bájtot igényel 3 bájt helyett.
A szerző 1978-ban javasolta először ezt a technikát, amely az opcodeokat és az indexeket egyetlen bájton belül kombinálja, majd elosztja a 256 elérhető bájtot is használatuk gyakoriságának megfelelően, mégis több mint 2 évtizednek kellett eltelnie ahhoz, hogy elfogadottá váljon.(sikere legyen).

5.4 Címzés

Az opcode-ok megtervezése fontos része az ISA-nak. Mindamellett egy program az bitek jó részét, arra használta, hogy meghatározza az operanduszok származási helyét, ahelyett, hogy műveleteket hajtott volna végre rajtuk.
Figyelembe kell venni az ADD utasítást is amelynek 3 operanduszra megadására van szüksége: két forrásra, és egy célra. (Ez a általános használatban egy időtartam operandus, ami mind a hármat kijelöli,a célterület pedig egy hely, ahol az eredmények tárolódnak.Az ADD-nek valahogyan meg kell mondania, hogy hol találhatók az operanduszok, illetve hogy hová tehető
(hol tárolható) az eredmény. Ha a memória-címek 32 bitesek, akkor az utasítások egyszerű leírásához, az opcode-hoz csatolt három 32 bites címre van szükség.
Két általános módszer használható a leírások méretének csökkentésére.Az első, ha az operandusz többször került használatba, akkor bekerült egy regiszterbe.Egy regiszter használata egy változó miatt, duplán hasznos: egyrészt a hozzáférés gyorsabb, másrészt kevesebb bitet kell lefoglalni az operandusz meghatározásához. Ha 32 regiszter van, akkor valamelyik közülük, képes csupán 5 biten meghatározni azt. Ha az ADD képes volt arra, hogy csak a regisztert használja, akkor mindössze 15 bitre

volt szüksége ahhoz, hogy mind a három operanduszt meghatározza, ezzel szemben 96 bitre volt szüksége akkor, ha azok mind a memóriában voltak.
Természetesen a regiszterek használata sem problémamentes. Valójában, néha a regiszterek használata több problémával jár.
Ha az operandusz a memóriában volt, akkor először be kellett tölteni egy regiszterbe, és ez több leíró (meghatározó) bitet
igényelt, mintha egyszerűn csak meghatároztuk volna a helyét a memóriában.Először is egy LOAD utasításra van szükség ahhoz, hogy az operandusz bekerüljön, egy regiszterbe. Ez nem csak egy opcode-ot igényel, hanem szükség van még a teljes memóriacím, illetve a cél-regiszter meghatározására is.Tehát, ha az operandusz csak egyszer volt használatban, akkor nem érdemes elhelyezni egy regiszterben.
Szerencsére, az évek során megmutatkozott, hogy az operanduszok sokszor kerülnek ismételt használatba.
Következésképpen a legfrisseb (mai) egységeket (építményeket) nagy számú regiszterrel látják el, és a legtöbb nagy terjedelmű helyi változó ezekben a regiszterkben tárolódik, így kiküszöbölhető a legtöbb memória-hivatkozás.Így tehát csökkent a program mérete, és futási ideje is.
A második eljárás csökkenti a meghatározás méretét és meghatározza egy vagy több operandusz értelmét (jelentését).
Számos technika létezik, ennek megvalósítására.Az egyik mód a önálló meghatározás használatára, egy forrás, illetve egy cél operandusz megadása.Az általános három címes ADD utasítás ezt a formát használja:
Céloperandusz = Forrás-Operandusz 1 + Forrás-Operandusz 2
A két címes utasítás korlátozott formát használ:
Regiszter 2 = Regiszter2 + Forrás-Operandusz 1
Ez az utasítás bizonyos értelemben ártalmas (veszélyes) lehet a Regiszter 2 tartalmára, mindaddig, amyg nem kerül rögzítésre.
Ha az eredeti változóra később szükség lehet, akkor azt először el kell másolni egy másik regíszterbe. Megállapodás szerint a két címes utasítások (parancsok) rövidebbek, de kevésbbé elterjedtek. Különböző tervezők, különbözőeket választottak.
A Pentium II például két címes ISA szintű utasítást használ, ellenben például az Ultra SRAC II -vell, ami három címes utasítást használ.
Csökkentsük az operanduszok számát a mi ADD utasításunkban hároról, kettőre. A korai számítógépek egyetlen regiszterrel rendelkeztek, ezt akkumulátornak hívták.Az ADD utasítás például mindig hozzáad egy "emlékeztető szót "az akkumulátorhoz, így csak egy operanduszra van szüksége a meghatározáshoz (az emlékeztetőre). Ez a technika jól dolgozik az egyszerű számításoknál, de amikor összetett, közbülső eredményekre van szükség, akkor az akkumulátornak vissza kell írnia a memóriába, és így az elérés késik.
Most eljutottunk a három címes ADD-k tól a két, illetve egy (szimpla) címesekig. Mit hagytunk ki ? A nulla (0) címeket ?
Igen. A negyedik fejezetben láttuk hogy az IJVM stac-et használ. Az IJVM IADD-nek nincsenek címei.
Mindkettő, a forrás, illetve a cél is megtalálható benne.
Hamarosan részletesebben is látni fogjuk, a stack címzését.

***[334-337] javított!

234. ábra
4. fejezet A mikroprogramok szintje
4-17. ábra A Mic-1 mikroprogramja (1. rész)

cimke	műveletek	magyarázat
Main1	PC=PC+1; fetch; goto(MBR)	MBR tárolja a műveleti kódot; következő byte beolvasása; elküldés(dispatch)
Nop1	goto Main1	Nem tesz semmit.
Iadd1	MAR=SP=SP-1; rd	Beolvassa a verem legfelső alatti szavát.
Iadd2	H=TOS	H=a verem teteje
Iadd3	MDR=TOS=MDR+H; wr; goto Main1	Összeadja a két felső szót; beírja a verem tetejébe
Isub1	MAR=SP=SP-1; rd	Beolvassa a verem legfelső alatti szavát
Isub2	H=TOS	H=TOS (top of verem-a verem legfelső szava)
Isub3	MDR=TOS=MDR-H; wr; goto Main1	Elvégzi a kivonást; beírja a verem tetejébe
Iand1	MAR=SP=SP-1; rd	Beolvassa a verem legfelső alatti szavát
Iand2	H=TOS	H=TOS
Iand3	MDR=TOS=MDR AND H; wr; goto Main1	“És” művelet végrehajtása; beírja az új verem tetejébe
Ior1	MAR=SP=SP-1; rd	Beolvassa a verem legfelső alatti szavát
Ior2	H=TOS	H=verem teteje
Ior3	MDR=TOS=MDR OR H; wr; goto Main1	“Vagy” művelet végrehajtása; beírja a verem új tetejébe
Dup1	MAR=SP=SP+1	Növeli SP értékét és bemásolja MAR-ba
Dup2	MDR=TOS; wr; goto Main1	A verem új szavának beírása
Pop1	MAR=SP=SP-1;rd	Beolvassa a verem legfelső alatti szavát
Pop2		Vár, amíg a memóriából betöltődik az új TOS
Pop3	TOS=MDR; goto Main1	Az új szó beolvasása a TOS-ba
Swap1	MAR=SP-1; rd	MAR értékét SP-1-re állítja; a verem második szavát beolvassa
Swap2	MAR=SP	A MAR-t a legfelső szóra állítja
Swap3	H=MDR; wr	TOS-t elmenti H-ba; a második szót beírja TOS-ba
Swap4	MDR=TOS	A régi TOS-t bemásolja MDR-be
Swap5	MAR=SP-1; wr	MAR értékét SP-1-re állítja; a verem második szavaként írja be
Swap6	TOS=H; goto Main1	TOS felülírása (update)
Bipush1	SP=MAR=SP+1	MBR=a verembe illesztendő byte
Bipush2	PC=PC+1; fetch	PC értékét növeli; következő műveleti kód beolvasása
Bipush3	MDR=TOS=MBR; wr; goto Main1	A konstans előjeles kiterjesztése; beillesztése verembe
Iload1	H=LV	MBR tartalmazza az indexet; LV-t bemásolja H-ba
Iload2	MAR=MBRU+H; rd	MAR=a beillesztendő helyi változó címe
Iload3	MAR=SP=P+1	SP az új verem tetejére mutat; írás előkészítése
Iload4	PC=PC+1; fetch; wr	PC-t növeli; következő műv. kód beolvasása; verem-tető átírása
Iload5	TOS=MDR, goto Main1	TOS felülírása
Istore1	H=LV	MBR tartalmazza az indexet; LV-t bemásolja H-ba
Istore2	MAR=MBRU+H	MAR=azon helyi változó címe, ahova menteni akarunk
Istore3	MDR=TOS; wr	TOS bemásolása MDR-be; szó írása

Istore4	SP=MAR=SP-1; rd	Beolvassa a verem legfelső alatti szavát
Istore5	PC=PC+1; fetch	PC-t növeli; következő műveleti kód beolvasása
Istore6	TOS=MDR; goto Main1	TOS felülírása
Wide1	PC=PC+1; fetch; goto (MBR OR 0x100)	Többszörös elágazás felső bitbeállításával
Wide-iload1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Wide-iload2	H=MBRU<<8	H=az első index byte 8 bittel balra tolva
Wide-iload3	H=MBRU OR H	H=a helyi változó 16 bites indexe
Wide-iload4	MAR=LV+H; rd; goto iload3	MAR=a beillesztendő helyi változó címe
Wide-istore1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Wide-istore2	H=MBRU<<8	H=az első index byte 8 bittel balra tolva
Wide-istore3	H=MBRU OR H	H=a helyi változó 16 bites indexe
Wide-istore4	MAR=LV+H; goto istore3	MAR=a helyi változó címe, ahol tárolni akarunk
Ldc_w1	PC=PC+1; fetch	MBR tárolja az első index-byte-ot; beolvassa a másodikat
Ldc_w2	H=MBRU<<8	H=első index byte<<8
Ldc_w3	H=MBRU OR H	H=16 bites indexet a konstans-készletbe
Ldc_w4	MAR=H+CPP; rd; goto iload3	MAR=a konstans készlet címe

cimke	műveletek	magyarázat
iinc1	H=LV	MBR tárolja az indexet; LV-t bemásolja H-ba
iinc2	MAR=MBRU+H; rd	LV+index értéket bemásolha MAR-ba; változó beolvasása
iinc3	PC=PC+1; fetch	Konstans beolvasása
iinc4	H=MDR	Változó bemásolása H-ba
iinc5	PC=PC+1; fetch	Következő műveleti kód beolvasása
iinc6	MDR=MBR+H; wr; goto Main1	Összeget bemásolja MDR-be; változó felülírása
goto1	OPC=PC-1	A műveleti kód címének eltárolása
goto2	PC=PC+1; fetch	MBR=az eltolási cím első byte-ja; második byte beolvasása
goto3	H=MBR<<8	Emelés és a megjelölt első byte H-ba mentése
goto4	H=MBRU OR H	H=16 bites elágazás eltolási cím
goto5	PC=OPC+H; fetch	Eltolási cím hozzáadása OPC-hez
goto6	goto Main1	Várakozás a következő műveleti kód beolvasására
iflt1	MAR=SP=SP-1; rd	A verem legfelső alatti szavának beolvasása
iflt2	OPC=TOS	TOS ideiglenes tárolása OPC-ben
iflt3	TOS=MDR	Az új verem tetejének tárolása TOS-ba
iflt4	N=OPC; if(N) goto T; else goto F	N bitre történő elágazás (Branch on N bit)
ifeq1	MAR=SP=SP-1; rd	A verem legfelső alatti szavának beolvasása
ifeq2	OPC=TOS	TOS ideiglenes tárolása OPC-ben
ifeq3	TOS=MDR	Az új veremadat tárolása TOS-ba
ifeq4	Z=OPC; if(Z) goto T; else goto F	Z bitre történő elágazás (Branch on Z bit)
if_icmpeq1	MAR=SP=SP-1; rd	A verem legfelső alatti szavának beolvasása
if_icmpeq2	MAR=SP=SP-1	MAR beállítása az verem új tetejének beolvasására
if_icmpeq3	H=MDR; rd	A verem második szavának bemásolása H-ba
if_icmpeq4	OPC=TOS	TOS ideiglenes tárolása OPC-ben
if_icmpeq5	TOS=MDR	Az verem új tetejének tárolása TOS-ba
if_icmpeq6	Z=OPC-H; if (Z) goto T; else goto F	Ha a két felső szó megegyezik, menjen T-be, egyébként F-be
T	OPC=PC-1; fetch; goto goto2	A goto1-hez hasonló, a célzott címhez szükséges

F	PC=PC+1	Az eltolási cím első byte-jának kihagyása
F2	PC=PC+1; fetch	PC most a következő műveleti kódra mutat
F3	goto Main1	Várakozás a műveleti kód beolvasására
Invokevirtual1	PC=PC+1; fetch	MBR=1-es indexbyte; PC növelése; 2. byte beolvasása
Invokevirtual2	H=MBRU<<8	Emelés, és első byte tárolása H-ba
Invokevirtual3	H=MBRU OR H	H=a program-mutató eltolási címe CPP-ből
Invokevirtual4	MAR=CPP+H; rd	A metódus-mutató beolvasása a CPP körzetből
Invokevirtual5	OPC=PC+1	Visszatérési PC ideiglenes tárolása OPC-ben
Invokevirtual6	PC=MDR; fetch	PC az új programra mutat; paraméter számláló beolvasása
Invokevirtual7	PC=PC+1; fetch	A paraméter számláló második byte-jának beolvasása
Invokevirtual8	H=MBRU<<8	Emelés és első byte mentése H-ba
Invokevirtual9	H=MBRU OR H	H=a paraméterek száma
Invokevirtual10	PC=PC+1	# helyváltozók első byte-jának beolvasása (# locals)
Invokevirtual11	TOS=SP-H	TOS=OBJREF-1 címe
Invokevirtual12	TOS=MAR=TOS+1	TOS=OBJREF címe (új LV)
Invokevirtual13	PC=PC+1; fetch	# hely második byte-jának beolvasása (# locals)
Invokevirtual14	H=MBRU<<8	Emelés/léptetés és az első byte mentése H-ba
Invokevirtual15	H=MBRU OR H	H=# helyi változók (locals)
Invokevirtual16	MDR=SP+H+1; wr	OBJREF felülírása a link-mutatóval
Invokevirtual17	MAR=SP=SP+1	SP, MAR beállítása arra a helyre, ahova a régi PC-t mentjük
Invokevirtual18	MDR=LV; wr	A régi PC mentése a helyi változók fölé
Invokevirtual19	MAR=SP=SP+1	SP arra a helyre mutat, ahol a régi LV tároljuk
Invokevirtual20	MDR=LV; wr	Régi LV mentése a mentett PC fölé
Invokevirtual21	PC=PC+1; fetch	Az új metódus első műveleti kódjának beolvasása
Invokevirtual22	LV=TOS; goto Main1	LV-t beállítja, hogy LV területére mutasson

cimke	műveletek	magyarázat
ireturn1	MAR=SP=LV; rd	SP, MAR nullázása az új link-mutató beolvasásához
ireturn2		Várakozás az olvasásra
ireturn3	LV=MAR=MDR; rd	LV beállítása a link-mutatóra; régi PC beolvasása
ireturn4	MAR=LV+1	MAR beállítása a régi LV beolvasásához
ireturn5	PC=MDR; rd; fetch	PC visszaállítása; következő műveleti kód beolvasása
ireturn6	MAR=SP	MAR beállítása a TOS írására
ireturn7	LV=MDR	LV visszaállítása
ireturn8	MDR=TOS; wr; goto Main1	A visszaadott érték mentése az eredeti veremtetőbe

4-17. ábra A Mic-1 mikroprogramja (3/3)

Ha az MBR-ben lévő byte csupa 0, a NOP utasításhoz tartozó műveleti kód a következő, úgy a következő mikroutasítás nop1-gyel jelölt lesz, és a 0. rekeszből lesz beolvasva . Minthogy ez az utasítás nem tesz semmit, egyszerűen visszaugrik a fő ág kezdetére, ahol a ciklus megismétlődik, de egy új MBR-be olvasott műveleti kóddal.

Újra kihangsúlyozzuk, hogy a mikroutasítások a 4-17. ábrán a memóriában nem egymás után következnek, és hogy a Main1 nem a nullás vezérlő tár címen van (mert a nop1-nek kell a nullás címen lennie). A mikroassembler végzi el minden egyes mikroutasítás megfelelő helyre történő elhelyezését és rövid sorozatokba történő rendezését a NEXT_ADDRESS mező felhasználásával. Minden sorozat az általa interpretált IJVM műveleti kód numerikus értékének megfelelő címen kezdődik (pl. POP a 0x57-es címen indul), de a ciklus többi része bárhol lehet a vezérlés tárolóban, és nem feltétlenül egymást követő címeken.

Most vizsgáljuk meg a IJVM IADD utasítását. A ciklusból az iadd1-gyel jelölt mikroutasításokhoz kell jutnunk. Ez az utasítás az IADD-ra jellemző műveletettel kezdődik:

1. A TOS már létezik, de a verem legfelső alatti szavát be kell olvasni a memóriából.
2. A TOS-t hozzá kell adni a memóriából behozott legfelső alatti szóhoz.
3. Az eredményt, amelyet be kell írni a verembe, el kell tárolni a memóriába, minthogy a TOS regiszterbe is.

Az operandus memóriából való betöltéséhez szükséges a verem mutató csökkentése, és annak beírása a MAR-ba. Meg kell jegyeznünk, hogy ez a cím egyben az is, melyet a következő írásnál használunk. Továbbá mivel ez a hely lesz a verem teteje, SP-hez is hozzá kell rendelni ezt az értéket. Ezért egyetlen művelet meg tudja állapítani SP és MAR új értékét, csökkentheti SP-t, és azt mindkét regiszterbe beírhatja.

Ezek az első műveletben mennek végre, (iadd1) és az olvasási művelet elindulnak. Emellett MPC megkapja az értéket iadd1 NEXT_ADDRESS mezőjéről, amely iadd2 helye, bárhol is van. Ekkor iadd2 be lesz olvasva a vezérlő tárból. A második ciklusban, mialatt várunk az operandusnak a memóriából történő beolvasására, bemásoljuk a verem legfelső szavát a TOS-ból H-ba, ahol elérhető lesz az összeadáshoz, mikorra a beolvasás befejeződik.

A harmadik ciklus (iadd3) elején, MDR tartalmazza a memóriából beolvasott összeadandót. Ebben a ciklusban ez hozzáadódik a H tartalmához, és az eredmény visszaíródik MDR-be, valamint a TOS-ba. Egy írási művelet is elindul, amely a verem tetejét elmenti a memóriába. Ebben a ciklusban a GOTO-nak a hatása az, hogy hozzáfűzi a Main1 címét az MPC-hez, eljuttatva minket a következő utasítás végrehajtásának kezdőpontjához.

Ha a következő műveleti kód, amit most az MBR tárolódik, 0x64 (ISUB), a műveleteknek majdnem pontosan ugyanaz a sorrendje alakul ki ismét. A Main1 elindulása után a vezérlés áttevődik a 0x64-en lévő mikroutasításnak (isub1). Ezt a mikroutasítást az isub2, isub3 és végül ismét Main1 követi. Az egyetlen különbség az előző és a mostani sorrend között, hogy az isub3-ban a H tartalmát kivonjuk MDR-ből az összeadás helyett.

Az IAND értelmezése majdnem azonos az IADD, és ISUB-éval, kivéve, hogy a verem két felső szava bitenként "és"-elődik, összeadás vagy kivonás helyett. Hasonló dolog történik az IOR-ral.

Ha az IJVC műveleti kód értéke DUP, POP vagy SWAP, a vermet meg kell változtatni. A DUP utasítás egyszerűen megismétli a verem tetejét. Mivel e szó értéke már tárolva van a verem tetején, a művelet ugyanolyan egyszerű, mint SP növelése, úgy, hogy az új helyre mutasson, és elmenteni TOS-t erre a helyre. A POP utasítás is majdnem ilyen egyszerű, csak csökkenti SP-t, a verem legfelső szavának törléséhez. Azonban a verem tetejének megtartásához most szükséges a verem új tetejének beolvasása a memóriából, és annak TOS-ba való írása. Végül, a SWAP utasítás magába foglalja a memória két helyén lévő értékek cseréjét: a verem két legfelső szaváak cseréjét. Ez valamivel könnyebb, mivel a TOS már tartalmazza ezen értékek egyikét, ezért ezt nem kell a memóriából beolvasni. Az utasítást a későbbiekben részletesebben tárgyaljuk.

A BIPUSH instrukció ennél kicsit komplikáltabb, mivel az utasítás kódját egy egyszerű byte követi, ahogy az a 4-18. ábrán látható. A byte-ot előjeles egészként kell értelmezni. Ezt a byte-ot, amely már betöltődött a memóriába a Main1 során, előjelesen ki kell terjeszteni 32 bitesre, és be kell illeszteni a verem tetejébe. Ez a művelet ezért ki kell, hogy terjessze előjelesen az MBR-ben levő byte-ot 32 bitessé, és be kell másolja MDR-be. Végül SP értéke nő eggyel, és be lesz másolva MAR-ba, engedélyezve az operandus kiíratását a verem tetejébe. Ezalatt ennek az operandusnak a TOS-ba is be kell másolódnia. Tehát, a főprogramhoz való visszatérés előtt, a PC értékét növelni kell, hogy a következő műveleti kód elérhető legyen a Main1-ben.

4-18. ábra A BIPUSH utasítás formátuma

***[338-341]

Nem érkezett meg! Ilonka István

***[342-345]

5-23. ábra Veremhasználat, mely kiértékel egy fordított lengyel formulát.

A másik választás a PC-rokon címzés. Ebben az eljárásban, magában az utasításban lévő (jelölt) offset van hozzáadva a programszámlálóhoz, hogy a célcímhez jusson. Valójában ez egyszerűen indexelt módszer, melyben a PC-használja regiszterként.

5.4.10. Az utasításkódok és a címzési módok merőlegessége

Szoftverileg az utasításoknak és a címzéseknek szokásos felépítésűnek kellene lennie az utasításformátumok minimális számával együtt. Egy ilyen felépítés sokkal könnyebbé teszi a fordítóprogram számára a jó kód megadását. Minden műveleti kódnak engedélyeznie kellene mindenfajta címzési módot, ahol van értelme. Továbbá minden regiszternek elérhetőnek kellene lennie mindenfajta regisztermód számára (beleértve a keretindexet(FP), a veremmutatót(SP) és a programszámlálót(PC)).

Példaként vegyünk egy korrekt konstukciót a 3-címzéses gépre, és tekintsük az 5-24. ábrán a 32 bites utasításformátumot. 256-ig a műveleti kódok fenn vannak tartva. Az első formátumban minden utasításnak két forrás- és egy célregisztere van. Az összes aritmetikai és logikai utasítás ezt a formátumot használja.

A címzés végén fel nem használt 8 bitnyi terület további utasítások megkülönböztetésére használható. Például egy műveleti kódot kijelölhetünk minden lebegőpontos művelethez, amelyek közül kiválik egy plusz terület. Azonkívül, ha a 23. bitet kitűzzük, akkor a 2. formátumot használjuk, és a másoodik operandus többé nem regiszter, hanem egy 13 bites, ami közvetlen állandóként van jelölve. A LOAD és STORE parancsok is használhatják ezt a formulát a segédmemóriában, indexelt módban.

Néhány további parancs szükséges, mint a feltételes ágak, de ezek könnyen beilleszthetők a 3. formátumba.

5-24. ábra Egy 3-címzéses gép utasításformátumainak egyszerű konstrukciója

Például egy műveleti kódot kijelölünk minden (feltételes) ágra, eljárásmódhívásra, stb., 24 bitnyi helyet hagyva a PC-rokon offsetre. Feltéve, hogy ez az offset szavakban számolt, a kapacitás $\pm$32 Mb-ban lenne. Néhány más műveleti kódot fenntartanak a LOAD és STORE parancsoknak, amelyek a hosszú offseteket igénylik (3. formátum). Ezek nem lennének teljesen általánosak (például csak az R0-t lehet menteni és tárolni), de ritkán használják.

Most pedig tekintsünk egy tervezetet egy 2-címzéses gépre, amely egy memóriájában lévő szót tud használni bármelyik operandushoz (5-25. ábra). Egy ilyen gép képes egy memóriabeli szót hozzáadni egy regiszterhez, egy regisztert egy szóhoz, egy regisztert egy regiszterhez vagy egy szót egy másik szóhoz. Jelenleg a memóriaárak viszonylag magasak, így ez a kivitelezés jelenleg nem népszerű, de ha a jövőben a cache- vagy a memóriatechnológia fejlődése következtében lejjebb mennek az árak, akkor ez egy különösen könnyű és hatékony konstrukció összeállítását eredményezi. A PDP-11 és a VAX, amik nagyon sikeres gépek voltak, két évtizeden keresztül domináltak a minikomputer- világban, az alábbi hasonló konstrukciót használva.

5-25. ábra Egy 2-címzéses gép utasításformátumainak egyszerű konstrukciója

Ebben a kivitelezében újra van egy 8 bites műveleti kódunk, de most 12-12 bitünk van a forrás és a célállomás pontos meghatározására. Minden egyes operandus esetén 3 bit adja a módot, 5 bit a regisztert és 4 bit az offsetet. A módot meghatározó 3 bittel el tudjuk látni a közvetlen-, a direkt-, a regiszter-, a regiszterközvetett-, az indexelt- és a veremmódokat, és még maradt hely további két jövőbeli módszer számára is. Ez egy korrekt és szabályos kivitelezés, amit könnyű megszerkeszteni és elég rugalmas, főleg ha a programszámláló, a veremmutató és a helyi változómutató az elérhető általános regiszterek között van.

Az egyedüli probléma itt az, hogy a direkt címzéshez a címhez több bitre van szükségünk. A PDP-11 és a VAX adott egy extra szót az utasításnak minden egyes közvetlenül címzett operandus címéhez. A két elérhető címzési mód közül bármelyiket használhatjuk egy 32 bites offsettel rendelkező indexelt módra, követve a parancsot. Így a legrosszabb esetben, vegyünk mondjuk egy memóriából memóriába menő ADD parancsot, amelynek mindkét operandusa közvetlenül címzett, vagy a hosszú indexelt formát használja, így az utasítás 96 bit hosszú lenne, és 3 busz(ciklus)t foglalna (egyet a parancsra, kettőt az adatokra). Másrészt a legtöbb RISC tervezetnek legalább 96 vagy talán még több bitre van szüksége, hogy egy memóriában lévő tetszőleges szót hozzá tudjanak adni egy másik, szintén memóriabeli tetszőleges szóhoz, és legalább 4 buszt használnak.

Az 5-25. ábrához sok alternatíva lehetséges. Ebben a kivitelezésben végre lehet hajtani az állítást

i=j;

az egyik 32 bites utasításban, feltéve, ha az i és a j is az első 16 feletti variációk esetén át kell térnünk a 32 bites offsetekre. Az egyik lehetőség egy másik formátum lenne, aminek egyetlen 8 bites offsetje van két 4 bites helyett, plusz egy kikötés, mely szerint vagy a forrás vagy a célállomás használhatja, de a kettő egyszerre nem. A lehetőségek és a választások korlátlanok, és a gépek tervezőinek manipulálniuk kell a tényezőket, hogy jó eredmények szülessenek.

5.4.11. A Pentium II címmegadási módszerei

A Pentium II címmegadási módszerei nagyon szokatlanok és különbözőek, attól függően, hogy egy egyéni utasítás a 16 vagy a 32 bites módban van. Lent mellőzzük a 16 bites módot; a 32 bites mód épp elég kellemetlen. A támogatott módszerek, beleértve a közvetlen, direkt, regiszter, regiszterközvetett, indexelt és egy speciális módszert, tömbösítik az elemeket. A probléma az, hogy nem minden módra vonatkoznak a parancsok, és nem használható mindegyik regiszter akármilyen módban. Ez még nehezebbé teszi a fordítóprogram-írók munkáját, és rossz kódokhoz vezet.

Az 5-23. ábrán a MODE byte felügyeli a címmegadási módszereket. Az operandusok egyikét a MOD és az R/M mező kombinációja határozza meg pontosan. A másik mindig egy regiszter, melyet a REG mező értéke ad meg. Az 5-26. ábrán felsorolt 32 kombinációt pontosan meg lehet határozni a 2 bitnyi MOD és a 3 bitnyi R/M mező segítségével. Például, ha mindkét mező nulla, az operandust az EAX regiszterben lévő memóriacímből olvassa ki.

A 01 és 10 oszlopok tartalmazzák azokat a módszereket, melyekben a regiszter az utasítást követő 8 vagy 32 bites offsethez van hozzárendelve. Ha kiválasztunk egy 8 bites offsetet, először előjel segítségével kiegészítjük 32 bitre, mielőtt még hozzáadnánk. Például egy ADD parancs esetén, ahol R/M=011, MOD=01, és a 6 offset közül egy kiszámítja az EBX és a 6 összegét, és elolvassa a memóriabeli szót az egyik operandus címén. Az EBX nem módosult.

A MOD=11 oszlop két regiszter közötti választásra ad lehetőséget. A szavas utasításokhoz az elsőt, a byte-os utasításokhoz a második lehetőséget választja.

5-26. ábra A Pentium II 32 bites címmegadási módszerei. M[x] az x-re vonatkozó memóriabeli szó

Vegyük észre, hogy a tábla nem teljesen szokványos. Például nincs út az EBP-n keresztül a közvetettbe és az ESP-ből az offsetbe.

Néhány módszerben egy újabb byte, amit **SIB**-nek (**Skála, Index, Alap**) hívnak, a MODE byte-ot követi (ld. 5-13. ábra). A SIB byte pontosan meghatározza a skálatényezőt és két regisztert. Ha a SIB byte jelen van, az operandus címét úgy lehet kiszámítani, hogy az indexregisztert megszorozzuk (a skálától függően) 1-gyel, 2-vel, 4-gyel vagy 8-cal, hozzáadjuk az alapregiszterhez, és végül hozzáadhatjuk egy 8 vagy 32 bites eltoláshoz, a MOD-tól függően. Majdnem az összes regiszter használható indexként is, alapként is. A SIB módszer hasznos az elérési tömbök számára. Például tekintsük a Java állítást erre:
(i = 0; I < n; I++) a[i] = 0;
ahol *a* a jelenlegi eljárásban lokális tömb, amely 4 byte-os egészekből áll. Az EBP-t tipikusan arra használjuk, hogy annak a veremkeretnek az alapjára mutasson, mely a lokális variációkat és a tömböket tartalmazza, mint ahogy azt az 5-27. ábra is mutatja. A fordítóprogram az *i*-t az EAX-ban tarthatja. Hogy *a*[*i*]-t elérje, egy SIB módszert kell alkalmaznia, melynek operanduscíme a 4 X EAX, az EBP és a 8 összege volt. Ez az utasítás el tudja tárolni *a*[*i*]-t egyetlen utasításként.

Megéri ez a módszer a fáradtságot? Nehéz megmondani. Kétségtelenül, ha ezt az utasítást helyesen használjuk, “megtakaríthatunk” pár ciklust. Hogy milyen gyakran hasznájuk, az a fordítóprogramtól és az alkalmazástól függ. Az a probléma, hogy ez az utasítás bizonyos nagyságú területet foglal el a chip felületén, amelyet másképp használhattunk volna fel, ha ez az utasítás nem lenne jelen. Például az első szintű cache lehetett volna nagyob, vagy a chip lehetett volna kisebb, és ezzel talán kicsivel nagyobb órasebesség lett volna elérhető.

Ezek azok a választások, amelyekkel a tervezők folyamatosan szembekerülnek. Általában, a széles körű szimulációs működéseket előbb megtervezik, minthogy bármit is szilíciumba öntenének, de ezekhez a szimulációkhoz szükséges a munkamegterheltség milyenségéről alkotott helyes elképzelés. Biztosan nyerő fogadás, hogy a 8088 tervezői nem csatoltak Web böngészőt teszthalmazukhoz.

***[346-349] javított!

5.4.12 Az UltraSPARC II címzési módjai

Az UltraSPARC ISA-ban minden parancs közvetlen vagy regiszteres címzést használ, kivéve akkor, ha az memória címzés. A regiszter módnál az 5 bit egyszerűen meghatározza, hogy melyik regisztert kell használni. A közvetlen módnál egy (előjellel ellátott) 13 bit hosszú konstans szolgáltatja az adatot. Nincs más mód ami használható lenne a számtani, logikai és ehhez hasonló utasitásokban.
A memória címzésnek három féle parancsa van. A LOAD, a STORE és egy multiprocessoros szinkronizáló utatitás. A LOAD, a STORE parancsoknak két módja van, hogy megcimezzék a memóriát. Az első mód kiszámolja két regiszternek az összegét és azután ezen keresztül közvetetten irányít. A másik egy hagyományos indexelés 13 bit eltolással.

5.4.13 Az JVM címzési módjai

A JVM-nek nincs általános címzési módja, abban az értelemben, hogy egy néhány bit mondja meg, hogy kell kiszámolni a címet (mint ahogy a Pentium II tudja). Helyette minden parancsnak van egy specifikus címzési módja ami összefüggésben áll vele. Mivel hogy nincsenek látható regiszterei a regiszter és a regiszter indirekt címzési mód még csak nem is lehetséges. Egy kis száma a parancsoknak , mint a BIPUSH, közvetlen címzést használ. Egyetlen mód ami rendelkezésre áll, az index mód. Ezt használja az ILOAD, ISTORE, LDC_W, és jó néhány egyéb utasítás a relatív változók és a alapértelmezett regiszterek specifikálására helyett, leggyakrabban az LV és a CPP. Az elágazások szintén az index módot használják.

5.4.14 A cimzési mód tárgyalása

Mostanra elég alaposan átnéztünk néhány cimzési módot. Ezek közül egyet használ a Pentium II, UltraSPARC II és a JVM, ezek összesitett táblázata a 5-28. Mint ahogyan be is jelöltük, nem mindegyik mód használható az összes instrukcióval.
A gyakorlatban nincs is szükség sok cimzési módra egy effektiv ISA rendszernél. Mivel kezdetben csaknem az összes kódot, amit ezen a szinten irnak, forditók generálnak, a szerkezet cimzési módjainak legfontosabb aspektusa az, hogy a választásokból kevés legyen, és azok világosak legyenek, és mindezt olyan ráforditással (a kód méretére és a végrehajtási időre nézve), hogy mindezek könnyen számithatók lehessenek. Ez azt jelenti általánosan, hogy a gép egy extrém megoldást használ: vagy ajánljon fel minden lehetséges válsztást, vagy csak egyet. Bármi dolog e kettő között azt jelent, hogy a forditó olyan választásokkal néz szembe, amikhez lehet, nincsen meg a tudása vagy a kifinomultsága.
Ergo a legátláthatóbb szerkezeteknek általában csak kevés cimzési módja van, szigorú megszoritásokkal használatukra. A gyakorlatban tehát, ha azonnali, direkt regiszter és indexelt módunk van, az elég szinte minden feladat végrehajtásra. Tehát, minden regisztert (beleértve a helyi (változó értékü) pointert, verem mutatót és program számlálót) fel kéne használni mindenho, ha akár egy regiszterre is szükség van. Komlikáltabb cimzési módok csökkenthetik az instrukciók számát, de az számitási sorozatok bevezetésének a kárára, hiszen különböző számitások részeit nem lehet könnyen párhuzamusan végezni.

Mostanra befejeztük tanulmányainkat arrók, hogyan is lehet opcodeokat és cimeket, valamit különböző cimzési módokat használni. Ha új computerhez kerülünk, akkor nemcsak a a cimzési módok és instrukciók alapos megismerése szükséges, nem csak azt kell látni, mi elérhető, miért azok a válaztások lettek alkalmazva és le kell vonnunk az alternativ választások konzekvenciáit.

5.5. INSTRUKCIÓ-TIPUSOK

ISA szinten az instrukciókat feloszthatjuk kb. fél tucat csoportra, amelyek gépről gépre ugyanazok, ennek ellenére különbözö részletei lehetnek. Tulajdonképpen minden gépnek van néhány kihasználatlan insturkciója az előző modellekkel való kompatibilitás miatt, mert a tervezőknek egy fantasztikus ötlete volt, esetleg a kormány tette azt bele, cserébe támogatást nyújtott. Most gyorsan áttekintjük az általános kategóriákat, anélkül, hogy kimeritenénk őket.

5.5.1. Adatmozgatásinstrukciók

Az adat másolása egyik helyről a másikra kell, hogy az egyik legfontosabb instrukció legyen. A másolásnál egy új objektum a másolat eredeti identikus (jellemző) bittel olyan lesz mint az igaz. A "mozgás" szó itt mást jelent, mint a normál magyar nyelvben. Amikor azt mondjuk, hogy Martin elment New Yorkból Kaliforniába, ezen nem azt érjük, hogy egy Martin-másolat Kaliforniában van, és az igazi még mindig New Yorkban van. Amikor azt mondjuk, a memória 2000. helyéről egy másik regiszterbe került az adat, akkor az eredeti még mindig a 2000 helyen maradt. Az adat mozgatási instrukciót más néven "adat duplikálási ionstrukcio", ahol az adat mozgatási idő kötött.

Két oka van, hogy az adatot le lehet másolni az egyik helyről a másikra: Az egyik alapvető tulajdonság pont a változók értékének a kijelölése. Az

A = B

kiadásakor a a B változó értéke a A változó memória cimére másolódik, hiszen a programozó ezt kivánta. A második ok pedig, hogy adatot másoljunk, hogy tárolni és kezeleni hatékonyan tudjuk. Mint láttuk, sok instrukció csak akkor tud változókat használni, ha azok regiszterben vannak. Kezedttől fogva két lehetséges forrás van, ahonnan az adatokat vehetjük (memória vagy regiszter) illetve ahova azok kerülhetnek (memória és regiszter), ezek tehát négy különböző másolási müveletet határoznak meg. Vannak computerek, amelyek négy ilyen instrukcióval rendelkezenkm, valamelyek csupán egyet használak mindegyikre. A LOAD parancs másol memóriából regiszterbe, a STORE regiszterből memóriába, a MOVE regiszterből regiszterbe, de a memóriából memóriából nincsen külön, ilyen szinten definiált müvelet.

Az adatmozgatási instrukcióknak rendelkezni kell valamilyen módon a kezelendő adat hosszáról. Néhány ISA rendszereknél rendelkezésre állnak olyan instrukciók, amely a változó mozgatásáat az első bittől akár az egész memória méretéig képesek kezelni. Meghatározott word (fixed-word-length) hosszúságú gépeknél a mozgatandó mennyiség gyakran egy szó.Bármilyen hosszabb vagy rövidebb mozgatást már szoftveres úton kell megoldani, akár shifeteléssel (átugrás) vagy összeolvasztással (merging). Néhány ISA rendszer további lehetőseget nyújt, hogy kevesebbet mint egy szót (rendszerint a bájtok számának csökkentésével), vagy pedig több szót is lehet. Többszavas adatlánc (multiple-word) másolása veszélyes is lehet, különösen, ha a

szószám nagy, mert sok időt vehet igénybe és esetleg a közepén meg kell szakitani. A változtatató szóhosszúságú gépek rendelkeznek olyan instrukciókkal, amelyek csak a forrást és a célt határozzák meg pontosan, a hosszt nem. A mozgatást addig folytatódik, amig el nem éri a adatvége jelet az adatban.

5.5.2. Dyadic müveletek

A dyadicus müveletek azok, amelyek két operandumot használnak fel, hogy egy végeredményt adjanak. Minden ISA rendszerben vannak olyan istrukciók amelyek összeadást, kivonást hajtanak végre egész számokon. Az egész számok szorzása és osztása is majdnem standardok. Feltételezhetően szükségtelen, hogy ezek fontosságát magyarázzuk. A dyadicus parancsok másik csoporja a booleani (logikai) instrukciók. 16 funkció létezik a két változóra. Habár kevés, ha egyáltalán van olyan gép, amelyek van saját insturkciója mind a 16-ra. Rendszerint NOT, OR, AND áll rendelkezésre, néha az EXCLUSIVE OR, NOR, NAND.

Az AND fontos használata, amikor egy szóval bitenként dolgozunk. Például, egy 32bit wordhosszúságú gépben 4 8 bites karaktert tárolnak szavanként. Feltételzvén, hogy elválaszuk a második karakter a másik háromtól, hogy ki tudjuk nyomtatni, vagyis kell egy szó, ami azt a karaktert tartalmazza (a jobb legszélső 8 bit), ezt hivjuk **right justified** (jobbról igazolt)nak, a baloldali szélen 24 bit zérókkal.

Az AND utatitásnál maszkot használunk Ennek az a lényege, hogy a nem kivánatos bitek zéróra változnak, kimaszkosulnak (masked out), mint lejjebb látható.

Az eredmény azért lesz eltolva jobb irányba 16bittel, hogy izolálják a karaktert a szó jobb végétől.

Az OR fontossága abban rejlik, hogy a biteket egy szóba csomagoljuk, mivel ez a kinyerés ellentéte. Azért, hogy egy 32bites szóból a első jobboldali 8 bitet, a többi zavarása nélkül, kimaszkoljuk, az uj karakter OR-ozzuk, mint lennt látható.

Az AND müvelet hajik arra, hogy elmozditsa az 1eket, mert soha nincs több egy az eredményben, mint az operandumok (összeadandók) összességében. Az OR müvelet képes rá, hogy 1-et szúrjon be, mert mindig van legalább annyi 1-es az eredményben, mint az operandumban.

***[350-353]
Nem érkezett meg. Marhella Krisztián: h837036

***[354-357]

A Java-ban a használt kifejezés a **method** (eljárás). Amikor a művelet befejezte a feladatát, vissza kell mennjen a hívás utáni kijelentésre. Ezért, a visszatérési címet a a műveletnek kell továbbitani, vagy elmenteni valahova, úgy, hogy az elérhető legyen amikor a visszatérésre kerül sor.

A visszatérési cím a következő három hely egyikében helyezhető el: a memórijában, a regiszterben vagy a sztek-ben. Messze a legrosszabb megoldás az egyedülálló, rögzitett memórija rekeszbe való helyezés. Ebben az elrendezésben, ha a művelet egy másik műveletet hív, a második hívás az első hívás visszatérési címének az elvesztéséhez fog vezetni.

Némi javulást jelent az, ha a műveletet hívó parancs a visszatérési címet a művelet első szavába helyezi, amivel az első végrehajtható parancs a második szó lesz. A művelet így visszatérhet úgy, hogy közvetetten elágazódik az első szóra, vagy, ha a hardware egy elágazási OP-kódot helyez az első szóba a visszatérési cím mellé, akkor közvetetten ágazódhat a visszatérési címre. A művelet hívhat más műveleteket, mert minden műveletben van hely egy visszatérési címnek. Ha a művelet saját magát hívja, ez az elrendezés nem vállik be, mert az első visszatérési cím megsemmisül a második híváskor. A művelet azon képessége, hogy önmagát meghívja, nevezetesen **rekurzió**, rendkivül fontos mint a teoretikusok úgy a programózók részére is. Továbbá, ha az A művelet hívja a B-t, a B művelet pedig hívja a C-t, és a C művelet hívja az A-t (közvetet vagy százszorszép-fűzér rekurzió), ez az elrendezés szintén nem vállik be.

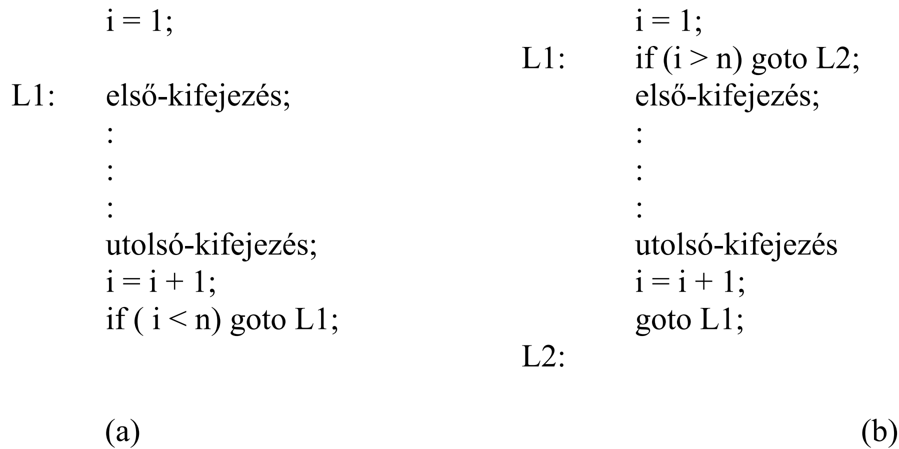
Nagyobb javulást jelent, ha a műveletet hívó parancs a visszatérési címet regiszterbe helyezi, amivel a biztos helyre való elhelyezés felelősegét a műveletre hagyja. Ha a művelet rekurziv, minden híváskor más-más helyre kell hogy tegye a visszatérési címet.

A legjobb dolog amit a műveletet hívó parancs tehet a visszatérési címmel, az az, hogy a sztekbe helyezze el. Amikor a művelet befejeződött, törli a visszatérési címet a sztekből, és betölti a program számlálójába. Ha a művelet hívás e formája lehetséges, a rekurzió nem okoz semmi különösebb gondot; a visszatérési címet automatikusan oly módon menti el, hogy mellőzi az előző visszatérési cím megsemmisítését.

5.5.6 Ciklus Vezérlés

Egyes parancscsoportok rögzített számszor való elvégzésére sűrűn keűl sor és ezért egyes gépeknek parancsaik vannak ennek megkönnyítésére. Az összes ilyen elrendezés magába foglal egy számlálot, amelyet növel vagy csökkent egy konstans ertékkel egyszer minden alkalommal amikor a cikluson végigmegy. Szintén, a számlálot egyszer ciklusonkent kivizsgálja. Ha bizonyos feltételek teljesülnek, a ciklus befejeződik.

Az egyik módszer a számlálót a cikluson kívül inicializálja és közvetlenül utána elkezdi a ciklus kódjának a végrehajtását. A ciklus utolsó parancsa megváltoztatja a számlálot és, ha a kilépési feltétel meg nem elégül ki, visszaágazódik a ciklus első parancsára. Különben, a ciklus befejeződik és továbblép, a ciklus utáni első parancsot hajtja végre. A ciklusszervezés e formáját végfeltételes tipusú ciklusnak nevezzük, és a 5-29(a) ábrán láthatjuk. (Itt nem tudtuk a Java-t igénybe venni, mert a Java-ban nincs goto parancs.)



```
        i = 1;                              i = 1;
                                    L1:     if (i > n) goto L2;
L1:     első-kifejezés;                     első-kifejezés;
        :                                   :
        :                                   :
        :                                   :
        utolsó-kifejezés;                   utolsó-kifejezés
        i = i + 1;                          i = i + 1;
        if ( i < n) goto L1;                goto L1;
                                    L2:

        (a)                                              (b)
```

Ábra 5-29. (a) végfeltételes ciklus. (b) kezdőfeltételes ciklus.

A végfeltételes ciklusnak az a tulajdonsága van, hogy a ciklus (ciklusmag) legalabb egyszer végrehajtódik, meg akkor is ha az n kisebb vagy egyenlő 0. Peldaként, vegyünk figyelembe egy programot, amely karbantartja a cégnek a személyzeti adatait. A program, egy bizonyos pontjan, olvas adatokat egy konkrét alkalmazottról. Ay n-be az alkalmazott gyermekeinek számát olvassa be, és végrehajtja a ciklust n-szer, egyszer gyermekként, olvasva a gyerek nevét, nemét és születési dátumát, hogy a cég tudjon küldeni neki egy születésnapi ajándékot, ami a cég egyik járulékos juttatása. Ha az alkalmazottnak nincs gyermeke, az *n* az *0* lesz, de a ciklus mégis egyszer végrehajtódik, amivel elküldi az ajándékokat és hibás eredményt ad.

Az 5-29(b) ábra egy másik módját mutatja a vizsgálatnak, amely az *n* kisebb vagy egyenlő *0*-ra is helyesen működik. Vegyük észre, hogy a vizsgálat különbözik a két esetben, ami a tervezőket arra kényszeríti, hogy vagy az egyik, vagy a másik eljárást válasszák.

Vegyük figyelembe a kódot, amely a kifejezés alapján jön létre

```
for (i = 0; i < 1; i++) {kifejezések}
```

Ha a kompájlernek nincs semmi információja az *n*-röl, kénytelen az 5-29(b) ábrán levő eljárást alkalmazni, hogy helyesen tudja kezelni az $n \leq 0$ esetet. Ha mégis, meg tudja határozni, hogy $n > 0$, például úgy, hogy látja hol van az *n* értékelve, akkor használhatja az 5-29(a) ábrán levő jobbik eljárást. A FORTRAN szabvány korábban azt állította, hogy minden alkalommal a ciklust egyszer végre kell hajtani, az 5-29(a) ábrán levő hatékonyabb kód létrehozása érdekében. 1977-ben ezt a hibát kijavították, amikor még a FORTRAN közösség is kezdte belátni, hogy nem volt jó ötlet az idegenszerű szemantikájú ciklusparancs, amely időnként rossz eredményt adott, még akkor sem, ha ciklusonként megsporolt egy elágazódási parancsot. A C és a Java ezt mindig helyesen végezték el.

5. 5. 7 Bemenet / Kimenet

Egyik másik parancscsoport sem létezik annyi változatban géptöl gépig, mint az I/O parancsok. Pillanatnyilag, három különböző I/O vázlatot használnak a személyszámítógépeken. Ezek:

1. Programozott I/O foglalt várással
2. Megszakítás-hajtású I/O
3. DMA I/O

Most sorjában e vázlatok mindegyikét megvitatjuk.

A legegyszerübb lehetséges I/O eljárás a programozott I/O, amelyet általában az olcsobb mikroprocesszerekben alkalmaznak, peldául, a beépített rendszerekben, vagy olyan rendszerekben amelyeknek gyorsan kell válaszolni a külső változásokra (real-time rendszerek). Ezeknek a CPU-knak altalában egy egyetlen kimenet (output) es egyetlen bemenet (input) parancsuk van. E parancsok mindegyike az I/O eszközök (készülékek) egyikét választja. Egyetlen betűt visz át a processzorban levő rögzített regiszter és a kiválasztott I/O eszköz között. A processzornak a parancsok egy meghatározott sorozatát kell elvégeznie minden írott vagy olvasott betűért.

E eljárás egyszerű peldájaként, tekintsünk meg egy terminált 4 1-bites regiszterrel, ami az a 5-30. ábrán látható. Két regiszter a bemenetre szolgál, egy státus és egy adat regiszter, kettő pedig a kimenetre, szintén egy státus és egy adat regiszter. Mindegyiküknek egyedülálló címe van. Ha memórija-térképezett I/O-t használunk, mind a négy regiszter része a számítógép memórija címterének és írható és olvasható a közönséges parancsokkal. Egyebként, különleges I/O parancsok, nevezetesen, IN és OUT, vannak biztosítva az írásra és olvasásra. Mindkét esetben, a bemenet és a kimenet is a processzor és a regiszterjei közötti adat és status információ átvitelével telyesül.

Betű rendelkezésre áll Következő betüre készen

Billentyűzet státus Képernyő státus

megszakítás engedélyezett megszakítás engedélyezett

Billentyűzet baffer Képernyő baffer
Kapott betű kimutatandó betű

Ábra 5-30. Az egyszerű terminál eszköz regiszterjei

A Billentyűzet státus regiszterben 2 bit használt, mig 6 bit nem. A legbalsóbb bitet (7) a hardwer 1-re állitja mindig, amikor betű érkezik. Ha a software elözőleg a 6-os bitet bekapcsolta, szakítás jön létre, másképpen nem (a szakításokat röviden később tárgyaljuk). Amikor programozott I/O-t használunk, hogy bemenetet kapjunk, a CPU rendes körülmények között rövid ciklusokat hajt végre ismételten olvasva a billentyűzet státus regiszterét, várva, hogy a 7-es bit következzen. Amikor ez bekövetkezik, a software beolvassa a billentyűzet baffer regisztert, hogy megkapja a betűt. A billentyűzet adat regiszter beolvasása okozza, hogy a BETŰ RENDELKEZÉSRE ÁLL bit újbol 0 értéket vesz fel.

A kimenet hasonloképpen működik. Ahoz, hogy betűt írjon a képernyőre, a software először a képernyő státus regisztert olvassa és vizsgálja, hogy a BETŰRE KÉSZEN bit értéke 1-e. Ha nem, a ciklus addig ismétlődik ameddig a bit nem lesz 1, azzal arra utalva, hogy az eszköz készen áll a betű elfogadására. Amint a terminál

készen lesz, a software beírja a betűt a képernyő baffer regiszterbe, ami azt okozza, hogy a betű képernyőre lesz továbbítva és a BETŰRE KÉSZEN bit a képernyő státus regiszterben törölve lesz. Amikor a betű ki lett írva és a terminál készen van a következő betű kezelésére, a BETŰRE KÉSZEN bitet a kontroller automatikusan újbol az 1 értékre állitja.

A programozott I/O példájaként, tekintsük meg az 5-31. ábrán látható műveletet. Ezt a műveletet két paraméterrel hívjuk: a betűsorral amelyet ki kell írni és a betűk számával a sorban, amely lefeljebb 1 K. A művelet teste (magja) egy ciklus amely egyenkent írja ki a betűket. Minden betűért, A CPU-nak várni kell amig az eszköz nem lesz készen, ekkor pedig kiírja a betűt. Az *in* és *out* olyan tipikus asszemblernyelv rutinok, amelyek olvassák illetve írják az első paraméterrel meghatározott eszköz regiszteréböl ill. regiszterébe a második paraméterként meghatározott változót. A 128-al való osztás megszabadít az alacsony-rendű 7 bittől, így 0 állapotban hagyva a BETŰRE KÉSZEN bitet.

```
public static void output_buffer(int buf[ ], int count) {
// Kiírja az adatok egy tömbjét az eszközön
 int status, i, ready;

  for (i = 0; i < count; i++) {
     do {
         status = in(display_status_reg);       //beolvassa a státust
         ready = (status << 7) & 0x01;          // elkülöníti a betűre készen bitet
     } while (ready = = 1);
     out(diplay_buffer_reg, buf[i]);
   }
}
```

Ábra 5-31. A programozott I/O peldája.

A programozott I/O elsődleges hátránya az az, hogy aCPU az idejének a javát rövid ciklusok ismétlésével tölti, mig várja, hogy az eszköz készen legyen. Ezt a hozzáállást **foglalt várásnak** nevezik. Ha a CPU-nak nincs más feladata (pl. a mosógépben levő CPU), a foglalt várás elfogadható lehet (habár még az egyszerű kontrollereknek is sűrűn kell kísérni több egyidejű eseményt). Akárhogyan, ha van más feladat is amit el kell végezni, mint amilyen a másik programok futtatása, a foglalt várás vázlat pazarló, ugyhogy más eljárásra van szükség.

A módja annak, hogy megszabaduljunk a foglalt várástol az, hogy a CPU elinditja az I/O eszközt és megmondja neki, hogy szakítást hozzon létre amikor elvégezte feladatát. Az 5-30. ábrán megmutattuk, hogy sebo@(anna)inf.u-szeged.hu

***[358-361]

Azzal, hogy az INTERRUPT ENABLE bitet egy gépi regiszterben helyezik el, a software kérheti, hogy a hardware adjon jelet, amikor az I/O befejeződött. A megszakítást részleteiben, ebben a fejezetben később fogjuk tanulmányozni, amikor áttérünk az irányításra.
Megéri megemlíteni, hogy sok számítógépben a megszakító jelet eredményezi, ha AND-eljük az INTERRUPT ENABLE bitet egy READY bittel. Ha a számítógép lehetővé teszi a megszakítást (mielőtt elindítja az I/O –t), hamarosan megszakítás fog bekövetkezni, mert a READY bit 1 lesz. Így szükséges lehet, hogy először elindítsuk a gépet, majd azonnal utána lehetővé tegyük a megszakítást. Egy byte beírása az állapot regiszterbe nem fogja megváltoztatni a READY bitet, amely csak olvasható.
Bár a megszakított-meghajtott I/O nagy lépés összehasonlítva a programozott I/O-val, de még távol áll a tökéletestől. Az a probléma, hogy a megszakítás minden közölt karakterhez szükséges. Kidolgozni egy megszakítást költséges. Az egyedüli lehetséges mód az, hogy meg kell szabadulni a megszakítások legtöbbjétől.
A megoldást akkor találjuk meg, ha visszamegyünk a programozott I/O-hoz, de valaki mással csináltatjuk ezt. (Nagyon sok problémára az a megoldás, ha valaki mással csináltatjuk meg.) Az 5-32 ábra mutatja, hogy kivitelezhető. Itt mi adtunk egy új chipet, egy DMA (Közvetlen Memória Bemenet) által irányítottat a rendszerhez, közvetlen hozzáférhetőséggel a bus-hoz.
A chipnek legalább négy regisztere van belül, amelyek közül mindegyik letölthető egy olyan software-rel, ami CPU-n fut. Az első tartalmazza az olvasható vagy írható memória címet. A második tartalmazza annak a számát, hogy mennyi byte-ot (vagy szót) fognak átvinni. A harmadik pontosan meghatározza a gép számát vagy az I/O címét, amelyet használni fognak, így pontosan meghatározva a kívánt I/O gépet. A negyedik azt mondja meg, hogy az adatot le fogják-e olvasni vagy le fogják-e írni az I/O gépre.
Ahhoz, hogy egy 32 byte-ból álló blokkot írjunk a 100-as című memóriából egy állomásra (mondjuk a 4-es gépre), a CPU az első három DMA regiszterbe írja a 32, 100 és 4 számokat, majd a WRITE kódját (mondjuk 1) írja be a negyedik regiszterbe, ahogyan azt 5-32 ábra mutatja. Miután beállítottuk ezeket, a DMA vezérlő arra kéri a bus-t, hogy olvassa le a százas byte-ot a memóriából, ugyanúgy ahogy a CPU olvasna a memóriából. Miután megkaptuk ezt a byte-ot, a DMA vezérlő kiad egy I/O parancsot a négyes gépnek, hogy írja le a byte-ot ennek. Miután mindezek a műveletek befejeződtek, a DMA vezérlő megnöveli a cím regisztert eggyel, és lecsökkenti a számregisztert eggyel.

Amikor a szám 0-ra ér a DMA vezérlő abbahagyja az adatok közlését, és követeli a megszakítást a CPU chipen. A DMA-val a CPU-nak csak néhány regisztert kell beállítania. Ezután szabadon megtehet bármit, amíg a teljes átvitel befejeződik, amely időpontban megszakítást kap a DMA vezérlőtől. Néhány DMA vezérlőnek 2 vagy 3 vagy, több regiszter sorozata van, így egyidejűleg többszörös átvitelt is tudnak irányítani.
Míg a DMA nagyon jól felszabadítja a CPU-t az I/O terhe alól, a folyamat nem teljesen szabad. Ha egy nagy sebességű eszköz, pl. egy disk a DMA-val kezd el futni, sok áttétel ciklus válik szükségessé mind a memóriához, mind az eszközhöz (a DMA-nak mindig gyorsabb az áttétele, mint a CPU-nak, mert az I/O eszközök nem tudják tolerálni a késést). Azt a folyamatot, amikor arra késztetjük a DMA vezérlőt, hogy ilyen bus ciklusokat vigyen a CPU-ról cikluslopásnak nevezzük. Mindazonáltal a nyereség azzal, hogy byte –onként (szavanként) egy megszakítást végzünk sokkal jobban, meghaladja a veszteséget, ami a cikluslopással jár.

5. 5. 8 a Pentium II parancsai

Ebben a fejezetben és a következő kettőben három gép, a Pentium II, az UltraSPARC és a pico JAVA II parancsait fogjuk áttekinteni. Mindhárom gépnek van egy olyan magja, amelyet a programozó mindig létrehoznak, valamint van olyan rész, amelyet ritkán használnak, vagy csak az operációs rendszer használ. Mi most a közös részekre helyezzük a hangsúlyt. Kezdjük a Pentium II-vel.
A Pentium II parancsrendszerében keverednek azok a parancsok, amelyeknek csak 32-bit módban van értelmük, és azok az elemek, amelyek a korábbi 8088 formához térnek vissza. Az 5-33-as ábrán az általánosabb egész számmal működő parancsok néhányát mutatjuk be, amelyeket a fordítok és a programozók napjainkban inkább, használnak. Ez a lista messze áll a teljestől, hiszen nem tartalmazza a lebegőpontos parancsokat, az irányító parancsokat, vagy még néhány megszokottól eltérőbb egész számmal működő parancsokat (például 8-bit byte használata AL-ben arra, hogy táblázatot készítsünk). Mindazonáltal jó áttekintést ad arról, hogy a Pentium II mit tud csinálni.
A Pentium II parancsai közül sok utal egy-két változóra, akár a regiszterben akár a memóriában. Például a kétváltozós ADD parancs a forrást, hozzárendeli a rendeltetési helyhez, és az egyváltozójú INC parancs növeli (egyet hozzáad) a változóhoz. Néhány parancsnak van sok rokon változata. Például a shift parancsok mind a bal, mind a jobb oldalra

el tud mozdítani és a jelet tudja akár speciálisan kezelni. A legtöbb parancsnak sok különböző kódja van, amely a változók természetétől függ.
Az 5-33-as ábrán az SRC mező az információ forrása és nem változik. Ezzel ellentétesen a DST mező a rendeltetési helyet jelenti, és amelyet a parancs valóban megváltoztat. Vannak arra vonatkozó szabályok, hogy mik azok, amelyek eredetként, illetve rendeltetési helyként szerepelhetnek, amelyek változhatnak szabálytalanul parancsról parancsra, de mi itt most nem megyünk ebbe bele. Sok parancsnak három változata van 8, 16 és 32 bit változókhoz. Ezek a parancsban található opcode-okban és/vagy bit-ekben különböznek egymástól. Az 5-33-as ábra a 32 bit parancsokat mutatja be.
A kényelem kedvéért a parancsokat több csoportba osztottuk. Az első csoport azokat a parancsokat tartalmazza, amelyek az adatokat mozgatják a gép körül, köztük a regiszter és a memória között. A második csoportba a számtani parancsok tartoznak, mind a jelzettek, mind a jelzetlenek. A szorzásnál és osztásnál a 64 bit-es eredmény vagy hányados EAX-ban (alacsony rendű rész) vagy EDX-ben (magas rendű rész) van elraktározva.
A harmadik csoportba a bináris kódolású tízes számrendszerbeli (BCD) számok tartoznak, amely minden byte-ot két négy bit-es részként kezel. Mindegyik rész egy tízes számrendszerbeli számot (0-9) tartalmaz. Az 1010-tól a 111-ig terjedő bit kombinációt nem használják. Így egy 16 bit egész szám állhat egy tízes számrendszerbeli számból 0-9999. Mivel ez a fajta tárolás eredménytelen, hatástalan, ezért kiküszöböli az arra irányuló szükségletet, hogy a bemenetet tízesről átalakítsuk kettősre, majd a kimenet miatt azt átalakítsuk tízesre. A parancsokat számtani műveletek elvégzésére használják a BCD számokon. Ezeket leginkább COBOL programoknál használják.
A Boolean és a shift/rotate parancsok irányítják a bit-eket vagy byte-okat számos módon. Számos kombináció létezik.
A következő két csoport a tesztelésre és összehasonlításra majd egyenesen az eredményre való ugrásra alkalmasak. A teszt és összehasonlító parancsok eredményei az EFLAGS regiszter számos bit-jében vannak elraktározva. A Jxx egy sor olyan parancsot helyettesít, amely a megelőző összehasonlítás eredményétől függően ugrik. (azaz bit-ek EFLAGS-ban)
A Pentium II-nek sok parancsa van a töltésre, tárolásra, összehasonlításra és karakterek vagy szavak sorának a beolvasására. Ezen parancsok, elé lehet illeszteni azt, hogy REP, amely addig ismétli ezeket, amíg egy bizonyos feltételt el nem érnek. Például az ECX minden ismétlés után csökken, amíg a 0-t el nem éri. Ezen a módon tetszőleges adat tömbök mozgathatók, összehasonlíthatók stb.

Az utolsó csoportba olyan parancsok keveréke tartozik, amelyek sehova máshova nem illenek. Ide tartozik az átalakítás, tömbök hosszának kezelése, a CPU leállítása és az I/O.
A Pentium II-nek számos előljárója van, amelyek közül egyet (REP) már említettünk. Az összes elöljáró olyan speciális byte, amely majdnem az összes parancsot megelőzhetik hasonlóan a WIPE-hoz IJVM-ben. A REP folyamatosan ismételteti a parancsot, amíg az ECX el nem éri a 0-t, mint ahogy előbb ezt már elmondtuk. A REPZ és a REPNZ folyamatosan végrehajtja az őt követő parancsot mindaddig, amíg a Z állapot beáll vagy nem áll be,

***[362-365]

illetőleg.A LOCK lefoglalja a buszt az utasítás végrehajtásának idejére,hogy biztosítsa a multiprocesszor szinkronizációt. Más prefixeket arra használnak, hogy rákényszerítsék az utasításokat a 16-bites vagy a 32-bites módra, ami nemcsak a hosszát vátoztatja meg az operandusnak, hanem teljesen újradefiniálja a címzési módot. Végül a Pentium II-nek összetett szegmentációs sémája van, kód, adat, verem és extra szegmensekkel, amit a 8088-tól örökölt.Vannak olyan prefixek, amelyek ezeknek a speciális szegmenseknek az elérését biztosítják, de ezekkel nem kell foglalkoznunk. (szerencsére)

5.5.9 Az UltraSPARC II Utasításai

Az UltraSPARC II-nek az összes fordító által használt egész típusú utasítását felsoroltuk az 5-34-es táblázatban.A lebegőpontos utasítások nem szerepelnek benne, sem a vezérlő utasítások (pl. cache kezelés, rendszer alaphelyzetbeállítása), sem a nem felhasználói címzési utasítások vagy az elavult utasítások. A beállítás meglepően kicsi: az UltraSPARC II igazán egy csökkentett utasításkészletű számítógép.

Az 1, 2, 4 és 8 bájtos LOAD és STORE utasítások hatása nyilvánvaló. A 64-bitesnél kisseb számot is 64-bites regiszterbe tölti, és a szám lehet előjel kiterjesztett vagy nulla kiterjesztett. Mindkét utasításnak létezik ilyen változata.

Az aritmetikai csoprt a következő. Az olyan utasítások, amelyeknek a nevében szerepel a CC beállítják az NZVC feltételes jelzőbiteket. A többi ezt nem teszi. CISC gépeken a legtöbb utasítás beállítja a feltételes kódokat, de a RISC gépeken ez nemkívánatos, mert ez korlátozza a fordító szabadságát az utasítások felcserélésében, ha megpróbál feltölteni egy üres helyet. Ha az erdeti utasítások sorrendje A...B...C, ahol A beállítja a feltételes kódokat és B pedig teszteli, a fordtó nem tudja beilleszteni C-t A és B közé, ha C beállítja a feltételes kódokat. Ezért sok utasításnak van két vátozata, így a fordító normálisan tudja használni az egyiket, ami nem változtatja meg a feltételes kódokat,hacsak nem később akarja tesztelni. Szorzás, előjeles osztás és előjel nélküli osztás egyaránt támogatott.

Cimkézett aritmetika egy speciális önigazoló formája a 30-bites számoknak. Ez használható nyelveknél, mint Smalltalk és Prolog, amelyekben a vátozók nem típusosak a fordítás idején és a típusuk megváltoztathatók a futás idején. Cimkézett számokkal a fordító tud ADD utasítást létrehozni és a futás alatt dönti el, hogy egész ADD vagy lebegőpontos ADD-ra van szükség.

A shift csoport tartalmaz egy bal és két jobb shiftet, mindkettő 32-bit és 64-bit kiterjesztett verzióval. Az SLL-nek mind a 64-bite shiftelve van, ugyanis ez kompatibilis a régi szoftverrel. A shifteket legtöbbször bit manipulációra használják. A legtöbb CISC gépnek sok shift és forgatás utasítása van, de ezek legtöbbje teljesen használhatatlan. Csak néhány fordító író szán ezekre

a szabadidejéből.
A logikai utasításcsoport hasonló az aritmetikaihoz. Ez tartalmazza az AND, OR EXCLUSIVE OR, ANDN, ORN és XNORN.Az utóbbi három

---

	Betöltések		**Logikai**
LDSB ADDR, DST	Előjeles bájt betöltése (8 bits)	AND R1, S2, DST	Logikai ÉS
LDUB ADDR, DST	Előjel néküli bájt betöltése(8 bits)	ANDCC	Logikai ÉS és icc beállítás
LDSH ADDR, DST	Előjeles félszó betöltése (16 bits)	ANDN	Logikai NEMÉS
LDUH ADDR, DST	Előjel nélküli félszó betöltése (16 bits)	ANDNCC	Logikai NEMÉS és icc beállítás
LDSW ADDR, DST	Előjeles szó betöltése (32 bits)	OR R1, S2, DST	Logikai VAGY
LDUW ADDR, DST	Előjel nélküli szó betöltése (32 bits)	ORCC	Logikai VAGY és icc beállítás
LDX ADDR,DST	Kiterjesztett betöltése (64-bits)	ORN	Logikai NEMVAGY
		ORNCC	Logikai NEMVAGY és icc beállítás
	Tárolások	XOR R1, S2, DST	Logikai KIZÁRÓVAGY
STB SRC, ADDR	Bájt tárolása (8 bits)	XORCC	Logikai KIZÁRÓVAGY és icc beállítás
STH SRC, ADDR	Fálszó tárolása (16 bits)	XNOR	Logikai KIZÁRÓVAGY
STW SRC, ADDR	Szó tárolása (16 bits)	XNORCC	Logikai KIZÁRÓVAGY és icc beállítás
STX SRC, ADDR	Kiterjesztett tárolása (64 bits)		
			Irányítás átirányítása
	Aritmetika	BPcc ADDR	Elágazás jóslással
ADD R1, S2, DST	Összeadás	BPr SRC, ADDR	Regiszteres elágazás
ADDCC	Összeadás és icc beállítás	CALL ADDR	Call eljárás
ADDC	Összeadás átvitellel	RETURN ADDR	Visszatérés az eljárásból
ADDCCC	Összeadás átvitellel és icc beállítás	JMPL ADDR, DST	Ugrás és linkelés
SUB R1, S2, DST	Kivonás	SAVE R1, S2, DST	Regiszterablakok kiegészítése
SUBCC	Kivonás és icc beállítás	RESTORE	Regiszterablakok visszaállítása
SUBC	Kivonás átvitellel	Tcc CC, TRAP#	Feltételes csapda

SUBCCC	Kivonás átvitellel és icc beállítás	PREFETCH FCN	Előhozza az adatot a memóriából
MULX R1, S2, DST	Szorzás	LDSTUB ADDR, R	Elemi töltés/tárolás
SDIVX R1, S2, DST	Előjeles osztás	MEMBAR MASK	Memóriatároló
UDIVX R1, S2, DST	Előjel nélküli osztás		
TADCC R1, S2, DST	Címzett összeadás		**Vegyes**
		SETHI CON, DST	Bitek állítása 10-től 31-ig
	Eltolások/forgatások	MOVcc CC, S2, DST	Feltételes mozgatás
SLL R1, S2, DST	Logikai balrashiftelés(64 bits)	MOVr R1, S2, DST	Regiszterbe mozgatás
SLLX R1, S2, DST	Logikai kiterjesztett balrashiftelés (64)	NOP	Nincs művelet
SRL R1, S2, DST	Logikai jobbrashiftelés (32 bits)	POPC S1, DST	Populáció számláló
SRLX R1, S2, DST	Logik. kiterjesztett jobbrashiftelés(64)	RDCCR V, DST	Feltételes kód regiszter olvasása
SRA R1, S2, DST	Aritmetikai jobbrashiftelés (32 bits)	WRCCR R1, S2, V	Feltételes kód regiszter írása
SRAX R1, S2, DST	Aritmetikai kiterj. jobbrashiftelés (64)	RDPC V, DST	Beolvassa a programszámlálót

SRC = forrásregiszter	TRAP# = csapdaszám	CC = feltételes kódkészlet
DST = célregiszter	FCN = függvénykód	R = cél regiszter
R1 = forrás regiszeter	MASK = operáció típusa	
S2 = forrás: regiszter vagy közvetlen	CON = konstans	cc = feltétel
ADDR = memóriacím	V = regiszter kijelölő	r = LZ, LEZ, Z, NZ, GZ, GEZ

Figure 5-34. Az UltraSPARC II elsődleges egészutasításai.

bizonytalan értékű, de mivel egy lépésben végrehajthatók és nincs szükség kiegészítő hardverre, így sokat gyorsíthatnak. Még a RISC gép tervezők is sokszor engednek a csábításnak.

A következő utasításcsoport tartalmazza az átvezérlést. BPcc képviseli az olyan utasítások készletét, amelyek elágaznak a különböző feltételekben és meghatározza az utasításban, hogy vajon a fordító szerint kell-e az elágazás vagy nem. BPcc teszetli a regisztert és elágazik, ha feltételt talált.

Két módon lehet az eljárásokat meghívni. A CALL utasítást az 5-14-es táblázat 4-es formája szerint használja a 30-bites PC-s word offset-et. Ez az

érték elég, hogy
a hívó bármely irányban bármely 2 gigabyte-nál közelebb levő utasítást elérjen.
A CALL utasítás leteszi a visszatérési címet R15-be, ami R31-be kerül a hívás után.
A másik út az eljárások meghívására a JMPL használata, mely az 1a vagy
az 1b formátumot használja és megengedi, hogy a visszetérési címet
belerakjuk valamely regiszterbe. Ez a forma hasznos,
ha a cél címet a végrehajtás alatt számoltuk ki.
SAVE és RESTORE befolyásolja a regiszterablakot és a veremmutatót. Mindkettőt
csapdázza, ha a következő (előző) ablak nem elérhető.
Az utolsó csoport néhány vegyes utasítást tartalmaz. A SETHI azért kell
mert nincs lehetőség arra, hogy egy 32-bites közvetlen oprandust a regiszterbe
töltsünk.
Ezt úgy oldjuk meg, hogy a SETHI-t arra használjuk, hogy 10-től 31-ig
a biteket beállítjuk és az ezt követő utasítást használva
közvetlen alakra hozzuk a biteket.
A populáció számláló utasítás egy rejtély. Ez a szóban levő 1 biteket számolja.
Azt híresztelik, hogy jó bombarobbanások szimulására és, hogy a
Los Alamos Nemzetközi Laboratórium (a nagy költekező) előnyben részesíti
azokat a gépeket, amelyek
tartalmazzák ezt az utasítást. Az utolsó három utasítás speciális registerek írására
és olvasására szolgál.
A listáról hiányzó ismert CISC utasítások zöme szimulálható
G0-al vagy egy konstans operandussal (1b formátum) . Néhányat közülük
megadtunk az 5-35-ös táblázatban. Ezeket ismeri az UltraSPARC II-es assembler
és a fordítók is gyakran használják. Közülük sok használja azt, hogy
G0 nullára van huzalozva és így a benne való tárolásnak semmi hatása sincs..

5.5.10 A picoJava II Utasítások

Itt az idő, hogy foglalkozzunk a picoJava II ISA szintjével. Ez megvalósítja az összes
226-utasítás JVM utasításkészletét, azaz 115 extra utasítást a
C, C++ implementációhoz és az operációs rendszer számára. Elsődlegesen a
a JVM utasításokra koncentrálunk, habár ez csak egy része a JAVA fordító
által használt utasításoknak.
A JVM ISA-nak nincsenek a felhasználó által látható regiszterei vagy
bármely olyan tulajdonsága, mint a legtöbb CPU-nak.
(A picoJava II-nek van 64 on-chip regisztere a verem tetjén, de
ezek azonban a felhasználói programoknak láthatatlanok)
A legtöbb JVM utasítás szavakat tesz a verembe vagy a már veremben levő
szavakon hajt végre műveleteket és szavakat vesz ki a veremből.
A legtöbb JVM utasítást közvetlenül a picoJava II hardvere hajtja végre, de
néhányuk mikroprogramozott és van néhány csapda a szoftveres futtatáshoz.

Utasítás	Hogy kell csinálni
MOV SRC, DST	OR SRC, G0 és tároljuk az eredményt DST-ben
CMP SRC1, SRC2	SUBCC SRC2, SRC1 és tároljuk az eredményt G0-ban
TST SRC	ORCC SRC1, G0 és tároljuk az eredményt G0-ban

NOT DST	XNOR DST, G0
NEG DST	SUB DST, G0 és tároljuk az eredményt DST-ben
INC DST	ADD 1, DST (közvetlen operandus)
DEC DST	SUB 1, DST (közvetlen operandus)
CLR DST	OR G0, G0 és tároljuk DST-ben
NOP	SETHI G0, 0
RET	JMPL %l7+8, %G0

Firure 5-35. Néhány szimulált UltraSPARC II utasítás.

Tehát egy kicsi ISA-szintű futtatási rendszer szükséges a működéshez, de ez a futtatási rendszer jóval kissebb, mint az egész JVM interpreter és csak ritkán használt. Tartalmazza a kódot néhány komplex utasítás értelmezésére, ezek az osztálybetöltő, byte-kód ellenőrző, szál menedzser és a szemétgyűjtő.

A JVM-nek egy viszonylag kicsi és jól érthető utasításkészlete van. A JVM utasításkészletet az 5-36-os táblázatban soroltuk fel, kivéve néhányat a széles, gyors és rövid formájú utasítások közül. Ezeket leszámítva a táblázat teljes.

A JVM utasítások típusosak, különböző utasítások vannak ugyanazon művelet megvalósítására különböző adattípusokon. Például az ILOAD utasítás egy 32-bites egészt tesz a verembe, amíg az ALOAD egy 32-bites hivatkozást (pointert) tesz a verembe.

Ez az erős típusosság nem szükséges a helyes művelethez, mivel mindkét esetben az eredmény az, hogy valamilyen memóriacímről 32-bit kerül a verembe.

Ez az erős típusosság teszi képessé a JVM-et arra, hogy futás időben ellenőrizze a JVM bináris programokat, hogy azok nem szegik meg a biztonsági korlátokat
úgy, hogy alattomosan egy egészt pointerré alakítva tiltott memóriaterületeket érjen el.

Menjünk végig a JVM utasításokon. Az első utasítás a táblázatban a
typeLOAD IND8

Tulajdonképpen ez nem egy utasítás, hanem egy sablon utasítások generálására.

A JVM különösen szabályos, ezért ahelyett, hogy az utasításokon egyesével végigmennénk néháhy esetben csak szabályokat adunk meg rájuk. Például a fenti esetben a type helyén a következő négy betű állhat: I, L, F vagy D, amelyek megfelelnek
az integer, long, float (32-bites lebegőpontos) és double (64-bites lebegőpontos) típusoknak.

Theát itt valójában négy utasítást adtunk meg (ILOAD, LLOAD, FLOAD, DLOAD) mindegyik 8-bites indexet használ egy lokális vátozó meghatározására és az értékét a verembe teszi.

***[366-369]

***[370-373]

370. – 373. oldalig
Az UltraSPARC II az ISA tervezetben egy adott szintet jelképez. Egy teljes 64 bites ISA-val rendelkezik (128-as busszal). Sok regisztere van és egy olyan utasítás készlete, ami hangsúlyozza a 3 regiszteres műveleteket (kiegészítve) a LOAD és a STORE utasítások egy kis csoportjával. Az összes utasítás ugyanakkora, habár a formátumok egy kicsit kicsúsztak az ellenőrzés alól. Mindenek ellenére ez egy kerülőút nélküli hatékony megoldáshoz vezet. A legtöbb új tervezet az UltraSPARC II-höz hasonlít, kevesebb parancsformátummal.

A JVM egy más konstrukció. Eredetileg az ISA-t úgy tervezték, hogy az applettek (kis programok) elküldhetők legyenek az Interneten, és hatékonyan fordíthatók legyenek a célállomáson software-ré. Ráadásul egy egynyelvű tervezet! Ez verem használathoz és, rövid, de variálható hosszúságú utasítások használatához vezetett, nagyon nagy utasítás sűrűséggel (átlag csupán 1,8 byte/utsítás). Egy olyan hardware létrehozása, ami egyszerre egy JVM utasítást hajt végre megcímezve a memóriát utasításonként kétszer vagy háromszor egy teljesítménybeli katasztrófa lenne. Azonban azáltal, hogy 64 szavas vermet teszünk a chipre és azáltal, hogy utasításokat sűrítünk azért, hogy a teljes szekvenciákat alakítsunk át, modern 3 címes RISC utasításokká, a picoJava II. vesz egy igen nem hatékony ISA-t, és jó munkát végez vele. A döntő tényező itt az, hogy egy modern számítógép magva egy mélyen huzalozott 3 regiszteres load/store RISC motor. Az UltraSPARC egyszerűen feltárja a motort a felhasználónak. A Pentium II elrejti azáltal, hogy egy régebbi ISA-t vesz és feldarabolja a CISC utasításokat RISC mikroműveletekké, melyet hatékonyan végre lehet hajtani. A picoJava II szintén a magvába rejti a RISC motort. De azt úgy teszi, hogy összekombinál több ISA utasítást egy RISC utasítássá, ami pont az ellentéte annak, amit Pentium II tesz
Összefoglalásképp, ha Goldilocksnak* kellene futatnia néhány számítógépes utasítást, ott kint az erdőn, látná, hogy néhány túl nagy (a Pentium II-es utasításait fel kellene darabolni), hogy néhány épp jó (az UltraSPARC II utasításait eredetiként kerülnek végrehajtásra). Ez a 3 medve elmélet a számítástechnikában.
(* Goldilocks angol lány mesehős. Aki betéved a három medve lakásába és felpróbálgatja a ruhákat. Természetesen csak az egyik medve ruhája lesz rá pont jó, a többi nagy vagy túl kicsi.)

6. A vezérlés folyamata

A vezérlés folyamata egy olyan szekvenciára utal, amelyben az utasításokat dinamikusan hajtják végre, más szavakkal a program végrehajtásközben. Általában az elágazások és az eljárás hívások hiányában a sikeresen végrehajtott utasításokat pozícionáljuk és beolvassuk az egymást követő memória helyekről. Ezek az eljárás hívások okozzák a vezérlési folyamat megváltozását. Megszakítva az éppen folyó eljárást és elindítva a meghívott eljárást. Ezek a korrutinok rokonságban állnak az eljárással és hasonló változást okoznak a vezérlés folyamatában. Ezek nagyon hasznosak a párhuzamos folyamatok szimulálására. A csapdák és a megszakítások is a folyamatok megszakítását okozzák. Amikor speciális körülmények lépnek fel. A következő részben mindezen témát megvitatjuk.

5.6.1 a vezérlés szekvenciális folyamata és az elágazások

A legtöbb utasítás nem változtatja meg a vezérlés folyamatát. Miután egy utasítást végrehajtott a memóriában, az ezt követő instrukciót pozícionálja és beolvassa, majd végrehajtja. Mindenegyes utasítás után a programszámláló növekedni fog az utasítás hosszával. Ha egy olyan intervallumon keresztül vizsgáljuk, ami hosszú, és összehasonlítjuk az átlagos utasítási idővel a programszámláló megközelítőleg: az idő lineáris függvénye megnövelve az átlagos instrukció hossz és az átlagos instrukció idő hányadosával. Másképpen fogalmazva a dinamikus sorrend, amelyben a processzor valójában végre hajtja az utasításokat, ugyan az, mint az a sorrend, amelyben az instrukciók megjelennek a magban. Ha egy program elágazásokat tartalmaz, akkor ez az egyszerű reláció nem valósul meg a fent említett két sorrend között. A függvény nem lesz lineáris. Ha az elágazások jelen vannak a programszámláló nem lesz többé monoton növő idő függvény.

Ennek következtében nehézzé válik az utasítások szekvenciáját látni a programból. A programozóknak, ha gondjuk van azzal, hogy nyomon kövessék azt a szekvenciát, ahogyan a processzor végrehajtja az utasításokat, hajlamosak hibázni. Ez a megfigyelés ösztönözte Dijkstra-t (1968) hogy írjon egy akkor vitatott levelet, amelynek az volt a címe, hogy "GO TO Statement Considered Harmful" (GO TO utasítás károsnak tekinthető), amelyben azt javasolja, hogy mellőzük a goto utasításokat. Ez a levél volt a szülőatyja a strukturált programozásnak. Amelynek az egyik elve a goto utasítások helyettesítése egy strukturáltabb formájú vezérlési folyamattal, mint például a "while" sorozat. Természetesen ezek a programok is lefordíthatók kétszintű programokra, amelyek lehet, hogy tartalmaznak sok elágazást, mert az if, while és más magas szintű vezérlési struktúrák végrehajtása elágazásokhoz vezet.

6,2 Elágazások

A programok strukturálásának a legfontosabb technikája az eljárás. Egy bizonyos szemszögből egy eljáráshívás, megváltoztatja a végrehajtás folyamatát, mint egy elágazás, azonban abban különbözik tőle, hogy az eljáráshívás teljesítése után az eljárás visszaküldi a vezérlést az eljárást hívó utasítást követő utasításra. Azonban egy másik szemszögből egy eljárás testet egy magasabb szinten lévő új utasítás meghatározásaként lehet tekinteni. Ezen álláspont szerint az eljáráshívást egy egyedi utasításként kell elképzelni, még akkor is, ha az eljárás meglehetősen komplikált. Ahhoz, hogy megértsünk egy eljáráshívást tartalmazó kód darabot, csak azt szükséges tudni róla, hogy mit tesz és, hogy hogyan teszi. Egy különlegesen érdekes utasításfajta a rekurzív utasítás, amely egy olyan utasítás, amely magát hívja elő vagy közvetlen vagy, közvetve egy más eljárás láncon keresztül. A rekurzív eljárás tanulmányozása komoly betekintést ad számunkra arról, hogy az eljáráshívások, hogyan hajtódnak végre és milyen helyi változatai vannak valójában. Most egy példát mutatunk a rekurzív eljárásra: a "Hanoi-i torony" egy nagyon ősi probléma, amelynek egy egyszerű megoldása van a rekurzív eljárás alkalmazásával. Egy bizonyos Honai-i kolostorban van három arany rúd. Az első rúdon 64 darab koncentrikus arany lemez van. Minden egyes lemez közepén van egy lyuk a rúd számára és minden lemez egy kicsit kisebb átmérőjű annál, ami közvetlenül alatta van. A második és a harmadik rúd kezdetben még üres. Az ottani szerzetesek szorgalmasan rakták át a lemezeket az első rúdról a harmadikra, úgy, hogy egyszerre csak egy lemezt raktak, és nem történhetett

olyan, hogy egy nagyobb lemez került egy kisebbre, mikor végeztek, azt mondták, hogy jön a világ vége. Ha manuális tapasztalatot akarunk, vagyis ki akarjuk próbálni, akkor minden jól fog menni, ha műanyag és kevesebb lemezt használunk. Azonban, ha így megoldjuk a problémát semmi sem fog történni. Ahhoz, hogy hozzájussunk ehhez a világvége effektushoz 64 arany lemezre van szükség.

Először csak n=3 lemezre bemutatva az eljárás:
Annak a megoldása, hogy az n számú lemezt elmozdítsuk az első rúdról a harmadikra, a következő lépéseket kell tennünk:

0. Alaphelyzet
1. Az első, vagyis a legkisebb lemezt áttesszük az első rúdról a harmadikra.
2. Az első rúdról a második lemezt a második rúdra tesszük.
3. A harmadik rúdon található legkisebb lemezt ráhelyezzük a második rúdon találhatóra.
4. Áttesszük az első rúdról a harmadikra a legnagyobb lemezt
5. A második rúdról a felső (legkisebb) lemezt rátesszük az első rúdra
6. Majd végül a második és első rúdról a lemezeket rátesszük a harmadik rúdon lévő legnagyobb lemezre.

Hogy megoldást találjunk a problémára szükségünk van egy olyan eljárásra, amellyel az n számú lemezt el tudjuk mozdítani az i rúdról a j rúdra. Mikor ezt a műveletet meghívjuk és megnevezzük:

Tornyok (n, i, j,) utasítással

***[374-377]

... Az eljárás elôször teszteli, hogy a n egyenlô-e 1-el. Ha igen, akkor a megoldás triviális, csak át kell tenni az egy diszket i-rôl j-re. Ha n nem=1, akkor a megoldás 3 részbôl áll, amit meg kell vizsgálni, mindegyiket, egy rekurziv eljárás hívásával.

A teljes megoldás látható az 5-40. ábrán. A

towers(3,1,3)

hívás, hogy megoldja az 5-39. ábra problémáját 3 új hívást generál, amelyek

towers(2,1,2)
towers(1,1,3)
towers(2,2,3)

Az elsô és a harmadik újabb 3 hívást generál, mindegyiket a teljes hétbôl.

5-40. ábra. Eljárás a Hanoi Tornyai megoldására.

A rekurziv eljárásokhoz, szükségünk van egy veremre, ahova eltároljuk a paramétereket és a lokális változókat minden híváskor úgy, mint az IJVM-nél. Minden alkalommal, amikor egy eljárás meghívunk egy új verem terület foglalódik le az eljárásnak a verem tetején. A legutóbb létrehozott veremterület az aktuális az aktuális veremterület. A példánkban a verem nô felfelé az alacsony memóriaterületekrôl a magasabbakba, csakúgy, mint az IJVM-nél.

A veremmutatóhoz való hozzáadáskor, amely a verem tetejére mutat, gyakran nem árt, ha van egy veremterület mutatónk, FP (Frame Pointer), amely egy állandó részre mutat a területen belül. Mutathat egy kapcsolatmutatóra, mint IJVM-nél, vagy az elsô lokális változóra. Az 5-41. ábra a gép veremterületét mutatja 32-bites szavakkal. Az eredeti towers hívás betette az n, i és j változókat a verembe, aztán elindított egy HIVO utasítást, ami betette a visszatérési címet a verembe, az 1012-es címre. A bejegyzésnél a meghívott eljárás törli a régi FP értéket (1000) az 1016-os címen és átírja a veremmutatót, hogy foglaljon tárat a lokális változóknak. Egy 32 bites változóval (k), SP (veremmutató) 4-el nô 1020-ra. Az 5-41. ábra mutatja az ezek után elôálló helyzetet.

5-41. ábra. A verem az 5-40. ábra szerinti mľködés néhány pontján.

Az elsô dolog, amit egy eljárásnak csinálnia kell, mikor meghívtuk, hogy elmentse az elôzô FP-t (így vissza lehet állítani az eljárás végén), átmásolja az SP- az FP-be, és talán növelje egy szóval, attól függôen, hogy az új terület FP-je hova mutat. A példában az FP az elsô lokális változóra mutat, de az IJVM-nél, LV a kapcsolatmutatóra mutatott. Különbözô gépek kissé különbözôen kezelik a veremterület mutatót, néha a veremterület aljára, néha a tetejére, néha pedig a közepére (mint az 5-41. ábrán.) téve azt. Ebben a megvalósításban hiba összehasonlítani az 5-41. és a 4-12. ábrát, hogy lássunk két különbözô módszert a kapcsolatmutató irányítására. Más módszerek is lehetségesek. Minden esetben a kulcs annak a lehetôsége, hogy végre tudjunk hajtani egy eljárást, vissza tudjunk térni és

vissza tudjuk állítani a verem
állását olyanra, amilyen volt, amikor még az határozta meg az aktuális
eljárás meghívását.

A kódot, ami elmenti a régi veremterület mutatót, kiszámolja az újat és átaszámolja a veremmutatót, hogy helyet foglaljon a lokális változóknak, eljárás elôzménynek (procedure prolog) nevezzük. Az eljárás végén a vermet újra fel kell szabadítani, amit az eljárás zárása (procedure epilog) végez. A legfontosabb számítógép jellemzôk egyike az, hogy milyen röviden és gyorsan tudja végrehajtani az elôzményt és a zárást. Ha ez hosszú és lassú, az eljáráshívás sokba fog kerülni. A programozók, akik imádják a hatékonyság oltárát, meg fogják tanulni elkerülni a sok rövid eljárás írását és óriási, monolitikus, strukturálatlan programokat írnak helyette. A Pentium II Enter és LEAVE utasításai úgy lettek megtervezve, hogy az eljárás elôzmények és zárások nagy részét hatékonyan végezzék. Természetesen van egy különleges modelljük, amiszerint a veremterület mutatót kezelhetik, és ha a fordító modellje más, nem tudják használni.

Most térjünk vissza a Hanoi Tornyai problémához. Minden eljáráshívás egy új területet ad a veremhez, és minden eljárás visszaadás töröl egy területet a verembôl. A verem használatának bemutatását egy rekurziv eljárás végrehajtsa során, a

towers(3,1,3)-mal

kezdôdô hívások felvázolásával folytatjuk. Az 5-41(a). ábra a vermet csak e hívás végrehajtása után mutatja. Az eljárás elôször teszteli, hogy n egyenlô-e 1, és annak felfedezése után, hogy n=3, beírja k-t és a következô hívást végzi

towers(2,1,2)

Miután ez a hívás megtörtént a verem az 5-41(b). ábra szerint alakul, és az eljárás ismét kezdôdik elejétôl (egy meghívott eljárás mindig az elején kezdôdik). Ekkor az n=1 vizsgálata megint hamis lesz, így újra beírja k-t és folytatja a következô hívással

towers(1,1,3)

A verem ekkor az 5-41(c). ábrán láthatóan néz ki, és a program számláló az eljárás elejére mutat. Az n=1 most teljesül és egy sor kiíródik. Ezután az eljárás visszatér kitörölve egy veremterületet és visszaállítva az FP-t és SP-t az 5-41(d). ábra szerint. Ez aztán folytatja a futást a visszatérési címen, amelyik a második hívás:

towers(1,1,2)

Ez új területet ad a veremhez, ami az 5-41(e). ábrán látható. Egy másik sor is kiíródik; a viisszatérés után egy terület törlôdik a verembôl. Az eljárások így folytatódnak, amíg az eredeti hívás végrehajtódik, és az 5-41(a). ábra által mutatott terület törlôdik a verembôl. A rekurzió mľködésének legjobb megértéséhez ajánlott végigjárni a

towres(3,1,3)

teljes futását papírral és ceruzával.

5.6.3 Társrutinok.

A szokásos meghívás sorozatnál van egy világos megkülönböztetés a meghívó eljárás és a meghívott eljárás között. Vizsgáljuk meg az A eljárást, ami meghívja a B-t az 5-42. ábrán.

A B eljárás számol valameddig, majd visszatér az A-ba. Elsô megvilágításban ajánlott a helyzetet szimmetrikusan átgondolni, mivel se A se B nem fôprogram, mindkettô eljárás. (Az A eljárás lehet fôprogramból meghívva, de ez most lényegtelen.) Továbbá, elôször az irányítás átadódik A-tól B-nek - a meghívás - késôbb pedig az irányítás átadódik B-tôl A-nak - visszatérés.

5-42. ábra. Mikor az eljárás meghívódik, annak futtatása mindig az elején kezdôdik.

Az asszimetria abból adódik, hogy mikor az irányítás átmegy A-ból B-be, a B eljárás futása az elejétôl kezdôdik; Mikor B visszatér A-ba, a futás nem az A elejétôl kezdôdik, hanem a meghívás helye utántól. Ha A fut valameddig és újra meghívja B-t, a végrehajtás ismét a B elején kezdôdik, nem az elôzô visszatérés helyén. Ha, a futás során A többször meghívja B-t, B végrehajtása mindig az elején kezdôdik újra és újra, mindenkor, ellenben A soha nem indul újra.

A különbségre a módszer világít rá, ami az irányítást váltja A és B között. Mikor A hívja B-t, akkor az eljárás meghívó utasítást használja, ami beteszi a visszatérési címet (azaz a hívást követô hely címét) valami használható helyre, például a verem tetejére. Ezután beteszi a B címét a programszámlálóba, hogy végrehajtsa a hívást. Mikor B visszatér, nem használja a hívó utasítást....

***[378-381]

Mikor B visszatér, nem használja a hívás utasítást hanem helyette a visszatérés utasítást használja amely egyszerűen kiveszi a visszatérési címet a veremből és beteszi a program számlálóba.

5-43. ábra: Mikor egy társrutin viszavette, a futtatás kezdetét veszi ott ahol előzőleg abbamaradt és nem az elején.

Néha hasznos ha két eljárás van, A és B mindkettő eljárásként hívja meg a másikat, mint ahogy az 5-43.-as ábra mutatja. Mikor B visszatér A-hoz, elágazik az állításhoz ami követi a hívást B-hez, mint fönt. Mikor A átengedi az irányítást B-nek, nem a kezdethez megy (kivéve az első alkalommal), hanem ahhoz az állapothoz ami a leggyakoribb "visszatérés", az az A leggyakoribb hívása. Két eljárás ami így működik a társrutin.

A társrutinok általános felhasználása az amikor párhuzamos futtatást szimulálnak egyetlen (egyedüli) CPU-n. Egyetlen társrutin sem fut valóban párhuzamosan a többivel, hacsak nincs saját CPU-ja. Ez a fajta programozás egyszerűbbé teszi néhány alkalmazás programozását. Hasznos olyan szoftverek tesztelésénél amelyek később több processzoron futnak majd.

Sem a szokásos hívás, sem a szokásos visszatérés utasítás nem használható a társrutinok hívására, mert a cím az ághoz a veremből jön. Mint egy visszatérés a társrutin saját magát hívja úgy, hogy tesz egy visszatérést valahova a rákövetkező visszatéréshez ami saját magához vezet.Az jó lenne, ha lenne egy utsítás ami kicseréli a verem tetejét a programszámlálóval. Részletesebben ez az utasítás először elővenné a régi visszatérési címet a veremből egy belső regiszterbe, majd betenné a program számlálót a verembe és végül a belső regisztert átmásolná a programszámlálóba. Mivel egy szó ki lett véve a veremből és egy szó be lett téve a verembe ezért a verem mutató nem változott. Ez az utasítás ritkán létezik ezért ezt szimulálni kell, mint jónéhány más utasítást a legtöbb esetben.

5.6.4 Csapdák

A csapda egyfajta automatikus eljárás hívás amit meghív néhány feltétel amit a program ad, többnyire egy fontos de ritkán előforduló feltétel. Jó példa erre a tólcsordulás. Sok számítógépen, ha az aritmetikai művelet túllépi a legnagyobb értéket akkor az ábrázolható. Egy csapda okozza, ami azt jelenti, hogy a vezérlés folyása átvált egy rögzített memória rekeszre, ahelyett hogy folytatná a sorrendet. Abban a rögzített rekeszben van egy ág egy eljáráshoz ami hívta **csapda kezelő**t, ami véghezvisz néhány helyénvaló cselekedetet. Mint például kiírja a hibaüzenetet. Ha egy művelet eredménye a tartományon belül van, nem okoz csapdát.

Nélkülözhetetlen a csapdáknál, hogy a program néhány kivételes feltétele okozza és a hardver vagy egy mikroprogram felfedezze. Egy másik módszer a túlcsordulás kezelésére, hogy létezik egy 1 bites regiszter, aminek 1-es értéke van, ha túlcsordulás történt. A programozó aki vizsgálni akarja a túlcsordulást, bele kell foglaljon egy egyértelmű "elágazik, ha a túlcsordulás bit be van állítva" utasítást minden aritmetikai művelet után. Ez a módszer lassú és helypazarló. A csapdák idő és memória takarékosak az egyértelmű programozó által felügyelt vizsgálathoz képest.

A csapdát megvalósíthatja egy egyértelmű mikroprogram (vagy hardver) teszt.

Ha túlcsordulást észlel a csapda címe betöltődik a programszámlálóba. Mi a csapda az első szinten talán a program felügyelet alatti szinten. Ha egy mikroprogram végzi a tesztet az időt takarít meg ahhoz képest mikor a programozó tesztel, mivel egy mikroprogram könnyen átlapozható. Memória takarékos is mivel csak egy helyen kell, hogy megtörténjen például a mikroprogram főciklusában függetlenül attól, hogy mennyi aritmetikai művelet történik a főprogramban.

Leggyakoribb, hogy a lebegőpontos számítások okoznak túlcsordulást, alulcsordulást, vagy egész számok túlcsordulnak, védelmi hibák, nem definiált opkódok, verem túlcsordulások, kísérletek nemlétező I/O eszköz elindítására, kísérletek egy szó feltöltésére páratlan címről, és nullával való osztásokkor.

5.6.5 Megszakítások

A megszakítások változások a vezérlés menetében, amit nem a futó program okoz, hanem valami más ami általában I/O-val van kapcsolatban. Például egy program utasítja a lemezt, hogy áthelyezzen információt, és állítsa be a lemezt, hogy hajtson végre egy megszakítást amint végez az áthelyezéssel. Mint a csapda a megszakítás is megállítja a futó programot és átadja az irányítást egy megszakítás kezelőnek, ami végrehajt néhány helyénvaló cselekményt. Mikor befejezi a megszakításkezelő visszaadja a vezérlést a megszakított programnak. Újra kell kezdenie a megszakított eljárást pontosan abban az állapotban ahol az volt mikor a megszakítás történt, ami azt jelenti, hogy minden belső regisztert vissz kell állítani az ő megszakítás előtti állapotába.

Az alapvető különbség a csapdák és megszakítások között az, hogy a csapdák szinkronban vannak a programmal, míg a megszakítások nem. Ha a program milliószor is tér vissza ugyanazzal a bemenettel a csapdák akkor is újratörténnek ugyanazon a helyen minden alkalommal, de a megszakítások váltakozhatnak attól függően például, hogy pontosan mikor üt le egy ember egy terminálnál kocsi visszát. Az ok a csapdák reprodukálhatóságára és a megszakítások nem reprodukálhatóságára az, hogy a csapdákat közvetlenül a program okozza, míg a megszakításokat - legjobb esetben - nem közvetlenül okozza a program.

Hogy lássuk valóban hogyan is működnek a megszakítások, hagyják hogy átgondoljuk egy általános példán keresztül: a számítógép kiakar tenni egy sornyi karaktert a terminálra. A rendszer szoftver először összegyűjti az összes karaktereket amit ki akar írni a terminálra egy bufferbe, inicializál egy globális változót a *ptr*-t, hogy a buffer elejére mutasson, és beállít egy második változót is *count* ami egyenlő a kiíratásra szánt karakterek számával. Ezután megvizsgálja, hogy a terminál készen áll-e, ha igen kiírja az első karaktert (Pl.: regisztereket használva, mint az 5-30-as ábrán). Elindítva az I/O-t, a CPU üresen készen áll, hogy elindítson más programot, vagy valami mást.

Természetesen kellő időben a karakterek megjelennek a képernyőn. A megszakítás most kezdődhet. Egyszerű formában a lépések folytatódnak.

HARDVER CSELEKMÉNY

1. Az eszközvezérlő érvényesít egy megszakítás vonalat a rendszer sínen, hogy elindítson egy megszakítás sorrendet.

2. Amint a CPU elkészült, hogy kezelje a megszakítást, ad egy nyugtázó

megszakítás jelzést a sínen.

3. Mikor az eszközvezérlő veszi, hogy az ő megszakítás jelzése nyugtázva van, tesz kis valós számot (integer) az adat vonalba, hogy azonosítsa magát. Ezt a számot nevezik a **megszakítás vektor**nak.

4. A CPU eltávolítja a megszakjtás vektort a sjnből és elmenti ideiglenesen.

5. Ekkor a CPU beteszi a program számlálót és a PSW-t (Program Állapot Szó) a verembe.

6. A CPU ekkor bemér egy új programszámlálót a megszakításvektor használatával, úgy hogy indexnek használja azt egy táblában a memória alján. Ha a program számláló 4 bájtos, pl. akkor a megszakítás vektor *n* megfelel a 4*n* címnek. Ez az új program számláló a megszakítást kezdeményező eszköz megszakítás szolgáltatás rutinjának elejére mutat. Gyakran a PSW-t is így töltik fel, vagy módosítják (pl.: hogy ne engedélyezzék a további megszakításokat).

SZOFTVER CSELEKMÉNY

7. Az első dolog amit a megszakítás szolgáltatás rutin tesz az az, hogy elmenti az összes regisztert, hogy később visszaállítható legyen. Elmenthetőek a veremben, vagy a rendszer táblában.

8. Minden megszakítás vektor rendszerint meg van osztva az adott típusú összes eszköz között, ezért nem lehet még tudni, hogy melyik terminál kérte a megszakítást. A terminál szám megtalálható pár eszköz regiszter olvasásával.

9. Bármely más megszakítás információ, mint az állapot kód, most olvasható.

10. Ha egy I/O hiba keletkezett, akkor azt itt kezeli le.

11. A globális változók, *ptr* és *count* fel lesznek újítva. Az előzőt növeljük, hogy a következő bájtra mutasson és a későbbi le lesz csökkentve, hogy megmutassa hogy egy bájtal kevesebb marad a kimeneten. Ha a *count* nagyobb, mint 0 akkor több karaktert kell megjeleníteni. Átmásolja a most a *ptr* által mutatott értéket az kimeneti buffer regiszterbe.

12. Ha szükséges egy különleges kódot bocsát ki, hogy tudassa az eszközzl, vagy a megszakítás vezérlővel, hogy a megszakítás végre lett hajtva.

13. Minden elmentett regisztert visszaállít.

14. Végrehajtja a VISSZATÉRÉS A MEGSZAKÍTÁSBÓL utasítást, visszatéve a CPU-t abba a módba és állapotba amiben a megszakítás történte előtt volt. A számítógép onnan folytatja ahol volt.

A kulcs fogalom a megszakításokkal kapcsolatban az **átláthatóság**. Mikor egy megszakítás kérődik, néhány esemény történik és néhány kód futódik, de mikor

minden befelyeződött a számítógép visszatérhet pontosan abba az állapotba amiben a megszakítás előtt volt. Egy megszakítás rutin ezekkel a tuljdonságokkal az átláthatóság. Ha minden megszakítás átlátható, akkor ez egyszerűbbé teszi a megértésüket.

Ha egy számítógépnek csak egy I/O eszköze van, akkor a megszakítások mindig úgy működnek ahogy leírtuk, és nincs semmi más amit elmondhatnánk róluk. Azonban egy nagy számítógépnek lehet több I/O eszköze is, és több is futhat egy időben, lehetséges, hogy több felhasználó parancsára. Ha egy nem nulla valószínűség fennáll mialatt egy megszakítás rutin fut egy második I/O eszköz igyekszik generálni egy megszakítást.

Két megközelítése van ennek a problémának. Az első egyetlen megszakítás rutint sem engedélyez a későbbi megszakításoknak, mint a legelső dolog amit tesznek, még mielőtt elmenti a regiszterket.

***[382-385]

Tannenbaum
Írta :Szikszai Sándor
H938348@sirius.inf.u-szeged.hu

??????...

Tehát, egy 9600 bps-es komonikációsvonalon, az adatok 1042 mikroszekundumonként érkeznek, akár kész van az egység, akár nincs. Ha az adat (karakter) feldolgozása nem fejeződik be, mikorra a következő karakter érkezik, adatvesztés léphet fel.

Amikor a számítógép időigényes I/O eszközöket kezel, a legjobb megoldás az, hogy az összes I/O eszközhöz prioritást rendelünk. Magasabbat az időigényesebbhez, alacsonyabbat a kevésbé kritikus eszközhöz.

Hasonlóan, a processzorhoz szintén rendelni kell prioritást, tipikusan ezt a PSW mező határozza meg. Az n prioritású eszköz megszakítást a kiszolgáló rutinjának is n prioritáson kell futnia.

Amíg az n prioritású megszakításrutin fut, addig más alacsonyabb prioritású eszközök megszakítását nem veszi figyelembe a processzor , amíg az n prioritású be nem fejeződik és a CPU vissza nem kapja a vezérlést.

Másrészt a magasabb prioritású eszközöknek nem szabad engedni, hogy sokáig tartson kiszolgálásuk.

A vezérlés megtartásának a legjobb módja az, hogyha megszakítások a észrevétlenül hajtódnak végre.

Most pedig, nézzük meg az összetett megszakításkezelést egy egyszerű példán keresztül.

Egy számítógépben 3 I/O eszköz található, egy nyomtató, egy lemezesegység és egy RS232-es komonikációsvonal, rendre 2, 4, és 5-ös prioritással, az adott sorrendben.

Kezdetben (a t=0 időpillanatban) a felhasználói program fut, amikor hirtelen a t=10. pillanatban a nyomtató megszakítást generál. A printer megszakításkiszolgálója az ISR elindul.(lásd 5.44-es ábra)

A t=15. időpillanatban az RS232-es is megszakítást kér. Mivel neki magasabb prioritása van a printerrel szemben, ez a megszakítás fog végrehajtódni elsőként. A gép az állapotváltozóit - melyet a printer használt- berakja a verembe (Stack), majd az RS232 kiszolgálórutinját elindítja. Egy kicsivel később (t=20), a lemezesegység befejezte az aktuális műveletét és kiszolgálást szeretne kérni. Azonban az ő prioritása kisebb, mint az éppen futó programé, a CPU nem fogadja el a megszakításkérést, függőben marad annak lekezelése.

A t=25. időpillanatban az RS232-es megszakítása befejeződik, és visszatér ahhoz az állapothoz ahol az előző megszakítást félbehagyta, nevezetesen a printer kiszolgálásához, ami 2-es prioritású. A CPU rögtön átkapcsol 2-es prioritási szintre, de mielőtt elindulna, a 4-es prioritású félbehagyott lemezmegszakítás aktiválódik, és csak ezután folytatódhat a printer lekezelése, ami t=40. pillanatban meg is történik. Végül a vezérlést visszakapja a felhasználói program.

A 8088-as óta az Intel processzorai két szintű megszakításkezeléssel rendelkeznek : a maszkolható és nem maszkolható megszakításokkal. A nem maszkolható megszakításokat kritikus hibák, illetve rendszerösszeomlások elhárítására használják, ilyenek például a memória paritáshibák. Minden I/O eszközhöz pedig maszkolható megszakítást rendeltek.

Amikor egy I/O eszköz megszakítást generál, a CPU a megszakításvektort használja arra, hogy a 256-bejegyzést tartalmazó táblázatból kikeresse a megszakításkezelő címét. A táblázat bejegyzései 8 bájtos szegmensdeszkriptorok, és a táblázat akárhol kezdődhet a memóriában. A globális regiszter ennek a kezdetére mutat. Egyetlen megszakításszint használatával nincs mód arra, hogy a CPU a magasabb prioritású eszközök megszakításait tiltsa az alacsonyabb prioritásúakkal szemben. Ezért az Intel processzorait külső meszakításvezérlővel látták el (8259-es mikrovezérlővel).

Amikor generálódik egy megszakítás, mondjuk n prioritással, a CPU megszakíja az addig folytatott munkáját. Ha a következő megszakítás magasabb prioritású, akkor a megszakításvezérlő másodszor is megszakítást generál.

Ha viszont a második alacsonyabb prioritású, akkor ennek a lekezelése csak az első megszakítás után kezdődik el.

A megszakításvezérlőnek tudnia kell az aktuális megszakításról, hogy az első befejeződött, ezért a CPU-nak parancsot kell küldenie, amikor az teljesen kész van.

5.7 Egy részletes példa : A Hanoi Tornyai

Most, hogy láttuk az ISA három gépét, nézzünk néhány példaprogramot mindháromra.
A példaprogram neve : Hanoi tornyai. A program Java-ban írt verziója az 5-40. ábrán található. Kövessük nyomon az Assembly -ben írt verziót is. Egy kicsit csalni fogunk, ahelyett hogy a Pentium II-re és UltraSparc II-re megírt Java változat fordítását adnánk meg, egy C - Assembly változatot fogunk megadni.
Az egyedüli különbség, hogy ami a Java-ban PrintLn, az a C verzióban
PrintF ("Move a disk from %d to %d\n",i,j) .
A mi céljainkhoz a PrintF szintaxisa teljesen lényegtelen, alapvetően a sztring szó szerint nyomtatásra kerül, annyi különbséggel, hogy a %D a soron következő egészet írja ki tízes számrendszerben.
Az egyetlen dolog ami lényeges, az eljárásnak három paramétert kell meghívnia : egy sztringet és két integert.
Az az oka annak, hogy a C verziót használjuk a Java verzió helyett , hogy a Java Input/Output könyvtár nem érhető el míg a C könyvtár elérhető. JVM-en mi is a Java verziót fogjuk használni.
A különbség minimális, itt csak a PrinF-re vonatozik .

5.7.1 A Hanoi tornyai Pentium II Assembly nyelven

Az 5-45.ábrán található a program egy lehetséges C-verziós fordítása Pentium II-re. A program első része az egyértelmű. Az EBP - t keret mutatónak, az első két Word-öt linkelésre használjuk, tehát az első aktuális paraméter,
az n (vagy N a MASM nem tesz megkülönböztetést) az EBP+8-nál van, majd ezt követi az i és a j az EBP+12 és EBP+16 -nál egyenként. Majd a lokális változó a k, az EBP+20-nál található. Az eljárás az új keret létrehozásával kezdődik, a régi tetejére rakja.
Ez úgy történik, hogy az ESP-t az EBP keretpointerbe másolja. Ezután n-t egyel hasonlítja össze, ha n>1 az ELSE-ágon folytatódik a program. Ha a THEN-ág teljesül, akkor a program a három változót a verembe helyezi : a formátumsztring kezdőcímét ,az i-t és a j-t , majd meghívja magát rekurzívan. A paramétereket fordított sorrendben helyezi a verembe, a C-program szabályai szerint. Emiatt a formátumsztring kezdőcíme a verem tetejére kerül.
Hogyha nem fordított sorrendben lennének a paraméterek, akkor a PrintF nem tudná milyen mélyen lenne a formátumsztring kezdőcíme.
A hívás után az ESP-t 12-vel növeli, hogy a paramétereket kiszedje a veremből. Természetesen a paraméterek nem törlődnek a memóriából. De az ESP megváltoztatása elérhetetlenné teszi őket a hagyományos veremműveletek révén.
Az ELSE elágazás, ami L1–nél kezdődik egyértelmű. Először is kiszámolja a 6–i–j értékét, amit a k változóban tárol. Az i és j értékétől függetlenül a harmadik rúd mindig 6–i–j. Ezt a k változóba elmentve nem kell újból kiszámítanunk.
Következőleg, az eljárás háromszor meghívja magát , mindig más paraméterekkel. Minden hívás után a verem törlődik és tartalma az eljáráshoz hozzáadódik. A rekurzív eljárások először megzavarhatják az embert, de egy adott szint után már egyértelműek. Mindössze annyi történik, hogy a paraméter értékeket a verembe rakja és utána az eljárás meghívja önmagát.

5.7.2 A Hanoi tornyai UltraSparc II Assembly nyelven

Most nézzük meg ezt a programot UltraSparc II-n is. A kód az 5-46.ábrán található. Mivel UltraSparc II kódja sokkal olvashatatlanabb, mint többi assembly kód, ezért használata gyakorlást igényel, de mi mégis megpróbálkozunk vele .....

***[386-387, 400-401]

386. oldal

Hogy ez működjön, a programot át kell küldeni a *cpp* nevű programon (C előfeldolgozó), mielőtt összeállítanánk. Mivel az UltraSPARC II assembler kisbetűket használ, mi is azokat használtunk itt(hátha valaki be akarja gépelni a programot és futtatni akarja).

Az UltraSPARC II és a PII verzió algoritmusa megegyezik. Mindkettő *n* vizsgálatával kezdődik, és az else ág végrehajtásával folytatódik, ha *n*>1. Az UltraSPARC II verzió bonyolultságát az ISA néhány sajátosságában kell keresni.

Először is az UltraSPARC II-nek át kell adnia a formátumstring címét a *printf*-nek, de a gép nem tudja csak egyszerűen berakni a címet a regiszterbe ami a kimenő paramétert tartja, mert nincs lehetőség arra, hogy berakjunk egy 32 bites konstanst egy regiszterbe egyetlen utasítással. Két utasításra van szükség, hogy ezt megtegyük: *SETH* és OR.

A következő megemlítendő dolog, hogy nincs szükség a verem beállítására a hívás után , mert az eljárás végén lévő RESTORE utasítás beállítja a regiszterablakot. A képesség, hogy a kimenő paramétereket regiszterbe töltsük és ne kelljen a memóriához fordulni nagy teljesítménynövekedést jelenthet, ha a hívások mélysége nem lesz túl nagy, de általánosságban elmondható, hogy az egész regiszterablak mechanizmus a bonyolultsága miatt nem éri meg a fáradságot.

Vegyük észre a NOP utasítást a *Done* után. Ez itt egy késleltető rés. Ez az utasítás mindig végrehajtódik, még akkor is, ha egy feltétel nélküli elágazás utasítást követ. A baj az, hogy az UltraSPARC II-nek mély pipeline-ja van és mire a gép felfedezi, hogy elágazáshoz érkezett, addigra a következő utasítást gyakorlatilag már végrehajtotta. Isten hozott a RISC programozás csodálatos világában.

Ez az érdekes tulajdonság hatással van az alprogram hívásokra is. Figyeljük meg a *towers* alprogram első hívását az Else ágon. *n*-1 –et tesz %o0-ba és a %o1-be, de már azelőtt meghívja a *towers*-t mielőtt az utolsó paraméter a helyére kerülne. PII-n először átadjuk a paramétereket, utána hívjuk az alprogramot. Itt először átadunk néhány paramétert, alprogramot hívunk, végezetül átadjuk az utolsó paramétert. Ugyanúgy, mint korábban ,mire a gép rájön, hogy egy *CALL* utasítással van dolga, addigra a következő utasítás olyan mélyen van a pipeline-ban, hogy azt már végre kell hajtani. Ebben az esetben miért ne használnánk a késleltető rést, hogy átadjuk az utolsó paramétert? Mégha a hívott alprogram legelső utasítása használná is azt a paramétert, akkor is időben helyére kerülne(a paraméter).

Végül a *Done*-nál láthatjuk, hogy a RET utasításnak szintén van egy késleltető rése. Ezt a RESTORE-nál használjuk, ami a CWP-t növeli, hogy visszarakja a regiszterablakot úgy, ahogy a hívó számít rá.

5.7.3 Hanoi tornyai JVM assemblyben

A kódot az 5-47 es ábrán láthatjuk. A megoldás viszonylag egyszerű, kivéve az I/O-t. Ezt a programot a Java fordító generálta, majd szerkesztve lett a

jobb olvashatóság
végett.

5-46 ábra megjegyzései:
/* az N a 0 input paraméter */
/* az I az 1 input paraméter */
/* a J a 2 input paraméter */
/* K a 0 lokális változó */
/* Param0 az output paraméter 0 */
/* Param1 az output paraméter 1 */
/* Param2 az output paraméter 2 */
/* cpp a megjegyzésekre a C szabályait használja*/
! if (n = = 1)
! if (n != 1) goto Else
!printf(”Move a disk from %d to%d\n”, i, j)
!Param0 = formátumstring címe
!Param1 = i
!meghívja printf-t mielőtt paraméter 2 (j) beállna
!a hívás után a késleltető rést használja a paraméter 2 beállításához
!készen vagyunk
!kitölti a késleltető rést
!k = 6 – i –j
!k = 6 –j
!k = 6- i – j
!towers(n –1 , i, k) indítása
! Scratch = N –1
!paraméter 1 = i
! meghívja towers-t mielőtt paraméter 2 (k) beállna
! a hívás után a késleltető rést használja a paraméter 2 beállításához
!towers(1, i, j) indítása
!paraméter 1 = i
! meghívja towers-t mielőtt paraméter 2 (j) beállna
!paraméter 2 = j
! towers(n-1, k, j)
!paraméter 1 = k
! meghívja towers-t mielőtt paraméter 2 (j) beállna
!paraméter 2 = j
!visszatérés
!késleltető rést használja a ret után, hogy visszaállítsa az ablakokat

16. Hány regisztere van annak a gépnek, amelyiknek utasítás formátumát az 5-24 ábra szemlélteti.
17. 5-24 ábrán a 23-as bitet az egyes és a kettes formátum megkülönböztetésére használjuk. Nincs azonban bit a hármas formátum használatának jelzésére. Honnan tudja a gép, hogy azt kell használnia?
18. Programozásban gyakori, hogy egy változóról el kell dönteni, hogy az A, B intervallum része. Ha egy három című utasításunk van A, B, X operandusokkal, akkor hány állapotkódbitet kellene ennek az utasításnak átállítania?
19. A PII-nek van egy olyan állapotkódbitje, amely követi a 3-as bit átvitelét egy aritmetikai művelet után. Mire jó ez?
20. UltraSPARC II –nek nincsen utasítása arra, hogy betöltsön egy 32 bites számot egy regiszterbe. Ehelyett általában két utasítást használunk: SETHI és ADD. Van még más mód, hogy betöltsünk egy 32 biten ábrázolt számot egy regiszterbe?
21. Egyik barátod éjjel 3-kor beront az irodádba, hogy elmondja vadiúj ötletét: egy utasítás két opkóddal. Elküldenéd a szabadalmi hivatalba vagy visszazavarnád a tervezőasztalhoz?
22. Az alábbi kifejezések vizsgálata megszokott a programozásban. Szerkessz egy utasítást, amely ezeket a vizsgálatokat hatékonyan elvégzi. Milyen mezők lesznek ebben az utasításban?

    if (n = = 0) ..
    if (i > j) ..
    If (k < 4) ..

23. Milyen hatással van az 1001 0101 1100 0011 16 bites bináris számra az alábbiak?
    a) eltolás jobbra 4 bittel, 0 kitöltéssel
    b) eltolás jobbra 4 bittel, jelkiterjesztéssel
    c) eltolás balra 4 bittel
    d) forgatás balra 4 bittel
    e) forgatás jobbra 4 bittel

16. Hogyan tudod egy memóriaszó tartalmát törölni egy olyan gépen, amelyen nincs CLR utasítás?
17. Számítsd ki az (A AND B) OR C boolean kifejezést, ha
18.

    A = 1101 0000 1010 1101
    B = 1111 1111 0000 1111
    C = 0000 0000 0010 0000

19. Hogyan lehetne egy A és B változó tartalmát kicserélni egy harmadik változó vagy regiszter használata nélkül? (*Segítség*: gondolj az EXCLUSIVE OR –ra)
20. Bizonyos számítógépeken lehetséges, hogy átrakjunk egy számot egy regiszterből egy másikba, eltoljuk őket balra más-más mennyiségekkel és összeadjuk az eredményeket kevesebb idő alatt, mint amennyire egy szorzásnak szüksége van. Milyen feltételek mellett lehet hasznos ez az utasítássorozat egy konstans és egy változó szorzatának kiszámítására?
21. Különböző gépeknek különböző az utasítássűrűségük(adott számítás elvégzéséhez szükséges byteok száma). Fordítsd le az alábbi Java kódrészleteket PII assemblybe, UltraSPARC II assemblybe és JVM-be. Utána számítsd ki hány

byteot használnak a különböző gépeken i és j lokális memóriaváltozók. Minden esetben a legoptimistább feltevéssel éljünk, ha valamilyen kérdés merülne fel!

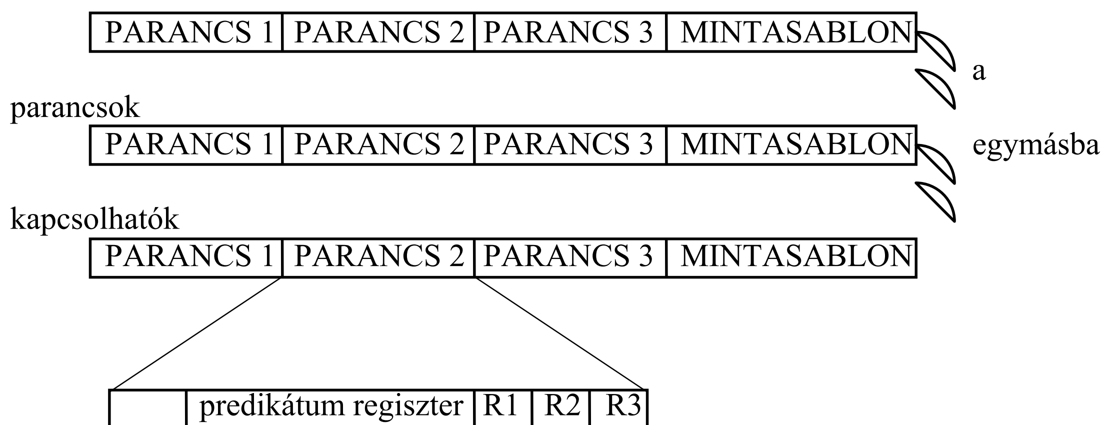
a) i = 3
b) i = j
c) i = j - 1

16. A szövegben megvitatott ciklusszervező utasítások általánosan a ciklusok kezelésére valók. Tervezz egy utasítást, ami hasznos lehet közönséges while (elöltesztelő) ciklusok kezelésére!
17. Tegyük fel, hogy a szerzetesek Hanoiban 1 lemezt tudnak percenként mozgatni.(nem sietnek a munka befejezésével, mert az ilyen tudással rendelkező embereknek korlátozottak a munkalehetőségeik Hanoiban) Mennyi ideig tartana nekik a 64 lemezes probléma megoldása? Az eredményt években add meg.
18. Miért helyezik az I/O eszközök a megszakításvektort a buszra? Lehetséges volna, hogy ehelyett ezt az információt egy táblában tároljuk a memóriában?
19. Egy számítógép DMA-t használ, hogy lemezről olvasson. A lemeznek 64 db 512 byteos szektora van sávonként. A lemez forgási ideje 16 msec. A busz 16 bit széles és egy buszátvitel 500 nsec-t igényel. Egy átlagod CPU utasítás 2 buszciklust igényel. Mennyire lassítja le a CPU-t a DMA?
20. Miért élveznek a megszakítás kiszolgáló rutinok elsőbbséget, miközben normál alprogramok nem kapnak semmilyen prioritást?
21. Az IA64 architektúra szokatlanul nagyszámú regisztert tartalmaz (64?). Összefüggésben állhat-e a regiszterek nagy száma melletti döntés az előrejelzések használatával? Ha igen milyen módon? Ha nem, akkor miért van belőlük annyi?
22. A szövegben már megvitattuk a spekulatív LOAD utasítás koncepcióját. Azonban nem történt említés spekulatív STORE utasításról. Miért nem? Megegyezik-e alapjában véve a spekulatív LOAD-dal, vagy van valamilyen más ok is, hogy nem foglalkoztunk velük?
23. Amikor két helyi hálózatot kell összekapcsolni, akkor egy bridge-nek nevezett számítógépet raknak közéjük, hozzákötve mindkettőhöz. Minden a hálózaton továbbított csomag generál egy megszakítást a bridge-n, hogy a bridge láthassa, hogy az adott csomagot továbbítani kell-e. Tegyük fel hogy, csomagonként 250 μsec szükséges a megszakítás lekezeléséhez és a csomag vizsgálatához, de –ha szükséges – a továbbítást a DMA végzi(hardwareből) a CPU leterhelése nélkül. Ha minden csomag 1K, akkor mennyi az elérhető maximális adatátvitel a hálózatokon anélkül, hogy a bridge csomagokat veszítene?
24. 5-41 ábrán a keretpointer az első lokális változóra mutat. Milyen információra van szüksége a programnak, hogy visszatérjen egy alprogramból?
25. Írj assemblyben egy alprogramot, ami egy előjeles bináris egészt (integer típusú) ASCII-be konvertál!
26. Írj assemblyben egy alprogramot, ami átkonvertál egy szóbelseji ragot ellentétes lengyel jelölésre. (?)

-401]

***[388-391]
Nem érkezett meg. Megyesi Zoltán: h938696

Ami rendkívüli, az az összefüggő parancskötegek fogalma. A parancsok hármas, kötegnek nevezett csoportokban jönnek, ahogy azt az 5-48 -as ábra mutatja. Mindegyik 128-bites bit köteg három, negyven bites állandó formátumú parancsot és egy nyolc bites mintasablont tartalmaz. A kötegek összeköthetők egy kötegvégi bit használatával, így több mint három parancs lehet jelen egy kötegben. A mintasablon információt tartalmaz arról, mely parancsok futtathatók párhuzamosan. Ez a séma és oly sok más regiszterek léte lehetővé teszi a fordítóprogram számára, hogy elkülönítse a parancstömböket és közölje a processzorral, hogy párhuzamosan futtathatók. Így a fordítóprogramnak újra kell rendezni a parancsokat, leellenőrizni az összefüggéseket, hogy megbizonyosodjon arról, hogy vannak elérhető funkcionális egységek, stb., a hardver helyett. Az ötlet az, hogy megvilágítva a gép belső működéseit és elmondva a fordítóprogramíróknak, hogy megbizonyosodjonak arról, hogy minden köteg kompatibilis parancsokból áll, a RISC parancsok táblázatba foglalása átvivődik a hardverről (futtatási idő) a fordítóprogramra (fordítási idő). Ez okból ezt a modellt EPIC-nek hívják (Explicity Paralell Instruction Computing).



5-48 -as ábra. Az IA-64 a három parancsból álló kötegekre alapszik.

Vannak előnyei parancsok fordítási időben való táblázatba foglalásának. Először is, mivel a fordítóprogram végzi az összes munkát, a hardver sokkal egyszerűbb lehet és így tranzisztorok millióit hagyhatja meg más hasznos funkciókra. Másodsorban minden adott programnál a táblázatba foglalást csak egyszer, fordítási időben kell elvégezni. Harmadszor, mivel a fordítóprogram végzi az összes munkát, a szoftverforgalmazó számára lehetővé válik, hogy egy olyan fordítóprogramot használjon, mely órákat tölt a program optimalizálásával és minden felhasználónak hasznos minden alkalommal amikor a program fut.

A parancskötegek ötlete felhasználható egész processzorcsaládok létrehozásához. Az alacsonyabbrendű processzorok órajelenként egy parancsköteget képesek feldolgozni. A processzor várhat míg az öszes parancs feldolgozása befejeződik, mielőtt megkezdené a következő köteget. A magasabbrendű processzorok hatványozott számú köteget tundának feldolgozni ugyanazon órajel alatt, analgikusan az aktuális szuperskaláris fejlesztésekhez. Szintén a magasabbrendű processzoroknál a processzor elkezdhet parancsokat kiadni egy új kötegből, mielőtt az előző köteg parancsait befejezte volna. Természetesen le kell ellenőriznie, hogy a szükségs regiszterek és a funkcionális egység elérhetőek-e, viszont nem kell leellenőriznie, hogy más parancsok abból a kötegből ütköznek-e vele, mivel a fordítóprogram biztosítja hogy erre ne kerüljön sor.

5.8.3 Predikáció

Másik fontos tulajdonsága az IA-64-nek, az új mód, ahogy kezeli a feltételes elágazásokat. Ha volna egy módszer, amellyel megszabadulhatnánk a többségüktől, sokkal egyszerűbb és gyorsabb processzrokat gyárthatnánk. Első látásra úgy tűnhet, hogy lehetetlen megszabadulni a feltételes elágazásoktól, mert a programok tele vannak if utasításokkal. Azonban az IA-64 egy predikáció nevű technikát alkalmaz, amely nagyon le tudja csökkenteni a számukat. Ezt most röviden leírjuk.

A mai gépeknél az összes parancs feltétel nélküli abban az értelemben, hogy amikor a processzor rátér egy parancsra, egyszerűen csak elvégzi azt. Nincs belső vitája arról, hogy "Csináljam vagy ne csináljam?". Ezzel ellentétben egy predikált architektúrában a parancsok feltételeket (predikációkat) tartalmaznak, melyek megmondják, mikor kell a parancsokat elvégezni és mikor nem. A paradigmaváltás a feltétel nélküli parancsokból predikált parancsokba az, ami lehetővé teszi, hogy megszabaduljunk a feltételes elágazásoktól. Ahelyett, hogy választanunk kellene a feltétel nélküli és a feltételes parancsszekvenciák között, az összes parancs egyesül egyetlen predikált parancsszekvenciába, mely más predikációkat használ más parancsoknál.

Hogy lássuk, hogyan működik a predikáció, kezdjük az 5-49 -es ábra egyszerű példájával, mely bemutatja a feltételes végrehajtást, a predikáció előfutárát. Az 5-49 (a) ábrán egy if utasítást látunk. Az 5-49 (b) ábrán ennek fordítását látjuk három parancsra: egy összehasonlítást, feltételes elágazást és egy move parancsot.

if (R1 ==0) CMP R1,0 CMOVZ R2, R3, R1

```
R2 = R3;           BNE L1
                   MOV R2, R3
                L1:
(a)                (b)                (c)
```

5-49 -es ábra. (a) If utasítás. (b) Az (a) assembly kódja. (c) Feltételes utasítás.

Az 5-49 (c) ábrán megszabadulunk a feltételes elágazástól egy új parancs, a CMOVZ használatával, ami egy feltételes move utasítás. Leellenőrzi, hogy az R1-es regiszter nulla-e. Ha igen, akkor az R3-at lemásolja R2-re. Ha nem, nem csinál semmit.

Mihelyt van egy olyan parancsunk, mely adatokat tud másolni mikor egy regiszter nulla, egy kis lépést tettunk egy újabb parancs felé, mely adatokat tud másolni, mikor a regiszter nem nulla, mondjuk a CMOVN. Rendelkezvén mindkét paranccsal úton vagyunk a teljes feltételes végrehajtás felé. Képzeljünk el egy if utasítást, néhány hozzárendeléssel a then részben, és néhánnyal az else részben. Az egész utasítás lefordítható egy kódra, hogy átállítson egy regisztert nullára, ha a feltétel hamis, és egy más értékre, ha igaz. Követve a regiszter setupját a then rész hozzárendelései lefordíthatók a CMOVN parancsok szekvenciájára és az else rész hozzárendelései pedig a CMOVZ parancsokéra.

Az összes ilyen parancs, a regiszter setup, a CMOVN és CMOVZ egyetlen alap tömböt alkotnak feltételes elágazás nélkül. A parancsok újra is rendezhetőek, vagy a fordítóprogram által (beleértve a hozzárendelések feloldását a teszt előtt) vagy a végrehajtás alatt. Az egyetlen trükk, hogy a feltételt már ismerni kell, mivel a feltételes parancsokat vissza kell vonni (a csatorna végének közelében). Egy egyszerű példa mutat be egy then és egy else részt az 5-50 -es ábrán.

```
if (R1 ==0) {            CMP R1,0        CMOVZ R2,R3,R1
      R2 = R3;           BNE L1          CMOVZ R4,R5,R1
      R4 = R5;           MOV R2,R3       CMOVN R6,R7,R1
} else {                 MOV R4,R5       CMOVN R8,R9,R1
      R6 = R7;           BR L2
      R8 = R9;       L1: MOV R6,R7
}                        MOV R8,R9
                     L2:
      (a)                (b)                 (c)
```

5-50 -es ábra. (a) If utasítás. (b) Az (a) assembly kódja. (c) Feltételes végrehajtás.

Habár csak nagyon egyszerű feltételes parancsokat mutattunk be (a Pentium II-ről), az IA-64 -en minden parancs predikált. Ez azt jelenti, hogy minden parancs végrehajtása feltételessé tehető. Az extra 6-bites mező (az előzőre vonatkozik) kiválaszt egyet a 64 1-bites predikációs regiszterekből. Így egy if utasítás le lesz

fordítva egy olyan kódra, mely beállít egyet a predikációs regiszterekből egyesre, ha a feltétel igaz, és nullára ha hamis. Ugyanakkor és automatikusan átállít egy másik predikációs regisztert az ellentétes értékre. A predikációt használva a gépi parancsok, melyek a then és else feltételt alkotják, egyetlen parancsba összegződnek, az előbbiek a predikáltat, az utóbbiak annak ellentétét használják.

Bár egyszerű, a példa az 5-51 -es ábrán bemutatja az alapfogalmát annak, hogyan használható a predikáció elágazások eltüntetésére. A CMPEQ parancs összehasonlít két regisztert és átállítja a P4-es predikációs regisztert 1-re ha egyenlőek, és nullára ha különböznek. Az if egy páros regisztert is (például P5) beállít az ellentétes állapotra. Most az is és a then rész parancsai egymás után rakhatók, mindegyik valamely predikációs regiszteren értelmezve (az ábrán zárójelezve). Tetszőleges kód helyezhető ide, ha minden parancs megfelelően predikált.


```
if (R1 ==R2)               CMP R1,R2        CMPEQ R1,R2,P4
        R3 = R4 + R5;      BNE L1           <P4> ADD R3,R4,R5
else                       MOV R3,R4        <P5> SUB R6,R4,R5
        R6 = R4 - R5       ADD R3,R5
                           BR L2
                       L1: MOV R6,R4
                           SUB R6,R5
                       L2:
        (a)                   (b)                (c)
```

5-51 -es ábra. (a) If utasítás. (b) Az (a) assembly kódja. (c) Predikált végrehajtás.

Az IA-64 -nél ezt az ötletet a végletekig vittük összehasonlító parancsokkal. A predikációs regiszterek beállításához és aritmetikai és más parancsokkal, melyek végrehajtása függ valamely predikációs regisztertől. A predikált parancsok sorban besűríthetők a parancscsatornába megszakítások és problémák nélkül. Ezért ilyen hasznosak.

A mód, ahogy a predikáció valójában működik az IA-64 -en az, hogy minden parancsot végrehajt. A parancscsatorna legvégén, amikor eljön az idő, hogy visszavonjon egy parancsot, leellenőrzi, hogy a predikáció igaz-e. Ha igaz, a parancsot hagyományosan vonja visza és az eredményeit visszaírja a célregiszterbe. Ha a predikáció hamis, nem történik visszaírás és a parancsnak nem lesz hatása. A predikációt a továbbiakban tárgyaljuk (Dulong, 1998).

5.8.4 Spekulációs Betöltések

Egy másik tulajdonsága az IA-64 -nek, mely felgyorsítja a végrehajtást, a spekulációs betöltések jelenléte. Ha egy betöltés spekulációs és sikertelen, egy kivétel okozása helyett csupán megáll és beállít egy bitet, mely a betöltendő regiszterrel függ

össze, mely a regisztert érvénytelenként jelzi. Ez az a "méreg-bit" melyet a 4-ik fejezetben mutattunk be. Ha kiderül, hogy a mérgezett regisztert később is használjuk, akkor rögtön fellép a kivétel, ellenkező esetben sohasem történik meg.

A spekulácót általában a fordítóprogram használja, hogy a helyükre emelje a betöltéseket használat előtt. Így korábban kezdve befejezhetők, mielőtt az eredményekre szükség lenne. Azon a helyen ahol a fordítóprogramnak szüksége van az éppen betöltött regiszterre, beszúr egy CHECK parancsot. Ha az érték jelen van, a CHECK NOP-ként viselkedik és a végrehajtás folytatódik. Ha az érték még nincs jelen, a következő parancsnak meg kell állnia. Ha egy kivétel lép fel és a "méreg-bit" aktív, a függőben lévő kivétel abban a pillanatban létrejön.

***[396-399]
Nem érkezett meg. Simon Nikoletta: h938818

16. Hány regisztere van annak a gépnek, amely parancsformátumai 5-24. ábra alatt vannak megadva?

17. 5-24 ábrákon 23 bitet használnak, hogy megkülönböztessék az 1. formátum használatát a 2.-étól. Azonban nincs bit arra, hogy a 3. formátum elkülönítéséről gondoskodjon. Hogyan tudja a hardware használni ezt?

18. A programozásban mindennapos, hogy a programnak szüksége van arra, hogy meghatározza, hogy hol van az X változó az A, B időközre vonatkozóan. Ha egy három-címes utasítás volt elérhető A, B és X mennyiségekkel, hány állapotbitet kellene elfoglalni ezen utasítás által.

18. A Pentium II-nek van egy állapotkódja, ami az aritmetikai eljárás után 3 biten tárolja a végrehajtás útvonalát. Mire jó ez?

20. Az Ultra SPARC II-nek nincs olyan parancsa, ami betölt egy 32 bites számot egy regiszterbe. Rendes körülmények között inkább két utasítás, a SETHI és ADD egyeztetését használják. Van-e több lehetőség arra, hogy egy 32 bites számot betöltsünk egy regiszterbe? Vitassuk meg a válaszokat.

21. Az egyik barátja éppen most rontott be a szobádba hajnali 3-kor levegőért kapkodva, hogy elmondja a ragyogó, új ötletét: egy utasítás két műveleti kóddal. A barátját a szabadalmi hivatalba vagy vissza a rajztáblához kellene küldenie?

22. Az if (n == 0)…
if (i > j) …
if (k < 4) … formulák tesztelése mindennapos a programozásban. Adjunk meg egy utasítást, ami ezeket a teszteléseket eredményesen végrehajtja. Melyik terület van jelen az utasításban?

23. Az 1001 0101 1100 0011 bináris 16 bites számon mutassuk meg a következményét a:
   a. egy 4 bites jobb áthelyezés 0 töltéssel
   b. egy 4 bites jobb áthelyezés a kiterjesztés jelölésével
   c. egy 4 bites bal áthelyezés
   d. egy 4 bites balra forgatás
   e. egy 4 bites jobbra forgatás

24. Hogyan lehet a memória szót kitörölni CLR parancs nélkül egy gépen ?

25. Számoljuk ki az (A AND B) OR C Boolean kifejezés értékét az
A = 1101000010101101

B = 1111111100001111

C = 0000000000100000 értékekre.

26. Adjon meg egy eljárást, mellyel két változó, A és B értékét felcserélhetjük
anélkül, hogy egy 3. változót vagy regisztert használnánk!
Ötlet: Gondoljunk a kizáró vagy-ra.

27. Bizonyos számítógépeken lehetőség van arra, hogy egy számot az egyik regiszterből a másikba rakjuk, mindegyiket áthelyezzük, hogy különböző mennyiségek maradjanak, és összeadjuk az eredményeket kevesebb idő alatt, mint ameddig egy szorzás tartana. Milyen esetekben hasznos ez az utasítássorozat "konstans * változó" kiszámítására?

28. Különböző gépeknek különböző utasítássűrűsége van (azon byteok száma, amelyek szükségesek egy bizonyos számítás végrehajtásához).
A következő Java kódú töredékeknek fordítsunk Pentium II assembly nyelvről, UltraSPARC II assembly nyelvről és JVM – ről. Azután számítsuk ki, hogy a gépeknek hány byte-ra van szükségük az egyes kifejezéseknél. Tegyük fel, hogy i és j helyi változók a memóriában, de egyébként minden egyes esetben a legpozitívabb feltételezést tételezzük fel.

A. i = 3
B. i = j
C. i = j – 1

29. A programciklus utasításokról úgy beszéltünk eddig, melyek a programciklusokat kezelték. Tervezzünk egy parancsot, ami hasznos lehet a megszokott programciklus kezelése helyett.

30. Tegyük fel, hogy Hanoiban a szerzetesek egy korongot tudnak megmozgatni percenként (nem sietnek a munka befejezésével, mert a munkalehetőségek Hanoiban korlátozottak ezzel a különös képességgel rendelkező emberek számára). Mennyi ideig fog tartani, míg a teljes 64 korongos problémát megoldják? Az eredményt években fejezzük ki.

31. Az I/O eszközök miért ismerik fel a félbeszakított vektort a bus-on? Lehetséges lenne azokat az információkat egy táblázatban tárolni a

memória helyett?

32. A számítógép a DMA – t használja, hogy a lemezről olvasson. A lemeznek sávonként 64512 byteos szektora van. A lemez forgási ideje 16 msec. A bus 16 bit széles és a busátvitel 500 nsec – ig tart. Az átlag CPU parancsának 2 bus ciklusra van szükségük. A DMA mennyire lassítja le a CPU – t?

33. A félbeszakított alkalmazásrutinoknak miért van elsőbbségük, míg a normál eljárásoknak nincs?

34. Az IA – 64 architektúra szokatlanul nagy számú regisztert tartalmaz (64)?. Azért van ilyen sok, mert összefüggésben van az állításokkal? Ha igen milyen módon? Ha nem, miért van olyan sok?

35. A szövegben, a LOAD utasítás elméleti fogalma vitatott. Azonban a STORE utasítás elméletét meg sem említik. Miért nem? Lényegében ugyanaz, mint a LOAD utasítás elmélete, vagy valami más oka van, hogy nem vitatják?

36. Amikor két helyi hálózat össze van kapcsolva, van egy számítógép, amit hídnak neveznek , és a kettő között van és mindkettővel össze van kapcsolva. Ha valamelyik hálózaton egy bitcsoport (csomag) mozog, a hídon szakadást okoz azért, hogy hagyja, hogy a híd észrevegye, ha a bitcsoportot továbbítani kell. Tegyük fel, hogy 250µsec-ig tart, hogy kezelje a félbeszakítást és ellenőrizze a bitcsoportot, de a továbbküldését, ha szükséges, a DMA hardware végzi anélkül, hogy megterhelné a CPU-t. Ha minden bitcsoport 1Kbyte-os, mi az a maximum adatmennyiség a hálózaton, amit még elvisel anélkül, hogy a híd bitcsoportot veszítene?

37. Az 5-41 ábrák alatt a sormutató az első helyi változót mutatja. Milyen információra van szüksége a programnak ahhoz, hogy visszatérjen az eljárásból?

38. Írjon egy assembly nyelvű subrutint, ami egy előjeles bináris egészet átír ASCII-be!

39. Írjunk egy assembly nyelvű subrutint, amely a beillesztett formulát átírja

a lengyel jelölésrendszerre.

40. A Hanoi-i torony nem az egyetlen kis rekurzív eljárás, amit a számítógépes szakemberek nagyon szeretnek. A másik állandó kedvenc téma az n!, ahol n!=n(n-1)! azzal a korlátozó feltétellel, hogy 0!=1. Írjon eljárást a kedvenc assembly nyelvén az n! kiszámítására!

41. Ha nincs meggyőződve arról, hogy a rekurzió néha nélkülözhetetlen, próbáljon meg programot írni a Hanoi-i toronyra anélkül, hogy rekurziót használna, vagy rekurziót szimulálna mátrix alkalmazásával. Azonban valószínűleg nem fog megoldást találni.

6.

Az operációs rendszer gépi szintje

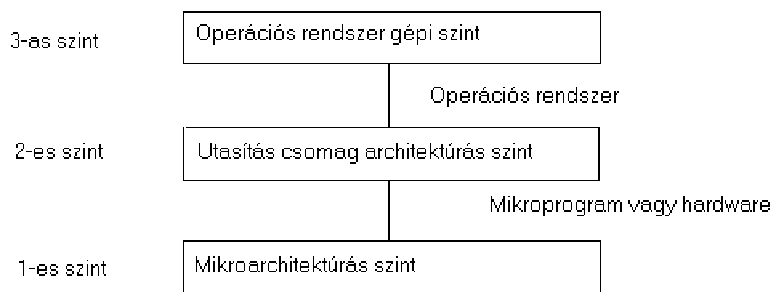
Ez a könyv a modern számítógép szintenkénti felépítéséről szól, ahol minden szint függ az előzőektől. Mi eddig már láttuk a digitális logika, a mikroarchitektura és az utasítások szintjét. Most itt az idő, hogy fentebb lépjünk egy szintet, az operációs rendszer birodalmába.

Az **operációs rendszer** egy program, amely a programozó nézőpontjából új utasítások és tulajdonságok választékát adja, előbb vagy utóbb azt, amit az ISA szint jelent. Általában az operációs rendszer végrehajtása nagyrészben a software-ban van, de nincs elméleti alapja, hogy miért nem tehető a hardware-ba, csakúgy, mint ahogyan a mikroprogramok vannak (amikor működnek). Röviden, azt a szintet, amely végrehajt, az OSM (Operating System Machine) szintjének fogjuk hívni. Ez a 6-1 ábrán van bemutatva.

Bár az OSM és az ISA szintje is absztrakt (abban az értelemben, hogy nem valódi hardware szintek), van egy fontos különbség közöttük. Az OSM szint utasításkészlete az a teljes utasításkészlet, mely a felhasználó programozóknak elérhető. Ez majdnem az összes ISA-szintű utasítást tartalmazza ugyanúgy, mint ahogyan az operációs rendszer az új utasítások választékát adja. Ezeket a parancsokat rendszerhívásnak (**system call**) hívjuk. A rendszerhívás az operációs rendszer egy előre definiált szolgáltatását hívja segítségül, amely valójában az egyik parancsa. Egy tipikus rendszerhívás adatokat olvas egy file-ból. Ki fogjuk szedni a kisbetűs Helvetica szóból a rendszerhívásokat.

Az OSM szint mindig értelmezett. Amikor egy aktuális program végrehajt egy OSM parancsot, mint például néhány adat file-ból való olvasása, az operációs rendszer lépésről lépésre végrehajtja ezeket az utasításokat ugyanúgy, mint ahogyan egy mikroprogram végrehajtana egy …

***[404-407]



6-1. Ábra Az operációs rendszer gépi szintjének elhelyezkedése.

ADD leírás, lépésről lépésre. Habár amikor a program végrehajt egy ISA szintű parancsot, ezt az alatta fekvő mikroarchitektúrás szint végzi el közvetlenül, az operációs rendszer mindennemű támogatása nélkül.

Ebben a könyvben az operációs rendszer tárgyának csak a legrövidebb bemutatását tudjuk nyújtani. Három fontos témára fogunk összpontosítani. Az első a virtuális memória, egy technika, amivel sok operációs rendszer rendelkezik, hogy úgy tűnjön, a gép több memóriával rendelkezik, mint amennyi ténylegesen van. A második a I/O fájl, egy magasabb szintű fogalom, mint a I/O parancs, amit már tanultunk az előző fejezetben. A harmadik és utolsó téma a párhuzamos feldolgozás - hogy képesek összetett folyamatok végrehajtani, kommunikálni, összehangolódni. A folyamat fogalma fontos, és le fogjuk írni később ebben a fejezetben. A létezés ideje alatt egy folyamatra úgy gondolhatunk, mint egy futó programra, és minden rendű információjára (memória, adattároló, programszámláló, I/O helyzet, és így tovább). Miután megtárgyaltuk ezeket az alapokat általánosságban, meg fogjuk mutatni, hogyan vonatkoznak két példa gépünk a Pentium II (Windows NT) és az Ultra SPARC II (UNIX) operációs rendszerére. Amióta a picoJava II - t rendes körülmények között beágyazott rendszereknek használják, nincsen teljes operációs rendszere.

6.1 VIRTUÁLIS MEMÓRIA

A számítógépek korai idejében a memóriák kicsik voltak és drágák. Az IBM 650 -es vezető tudományos számítógép volt ezekben a napokban (1950-es évek vége), csak 200 szavas memóriával rendelkezett. Az egyike a legelső ALGOL 60 fordítóprogram csak 1024 szavas memóriájú számítógépre készült. Egy korai időmegosztó rendszer egész jól futott egy PDP-1-en aminek a teljes memória mérete csak 4096 18-bit szó az operációs rendszernek és a felhasználói programoknak együttesen. Azokban a napokban a programozók sok időt töltöttek azzal, hogy próbálták beleszorítani a programokat a kevéske memóriába. Gyakran szükséges volt egy sokkal lassabban futó algoritmust használni, mint egy másik, jobb algoritmust, pusztán azért, mert a jobb algoritmus túl nagy volt - vagyis, a program a jobb algoritmust használó programmal előfordult, hogy nem lehetett belepréselni a gép memóriájába.

A tradicionális megoldása ennek a problémának a másodlagos memória használata volt, mint például a lemez. A programozók több részre osztották a programot beborításnak (OVERLAYS) hívták ezeket, amiknek mindegyike belefért a memóriába. Hogy fusson a program az első overlay-t vették, és ez futott egy darabig. Amikor végzett, akkor beolvasta a következőt, majd előhívta, és így tovább. A programozó felelős volt a programok overlay-ekre való darabolásáért, döntően abban, hogy a másodlagos memóriában melyik overlay-t hol tartják, az overlay-ek szállításának elrendezésében a fő memória és a másodlagos memória között, és általánosságban az egész overlay folyamat kezelésében a gép bármilyen segítsége nélkül.

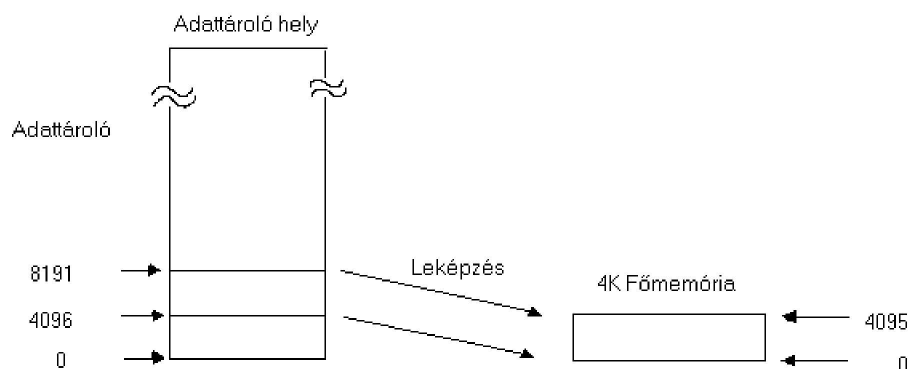
Habár széleskörűen használták sok éven át, ezzel a technikával az overlay-ek kezeléssel kapcsolatba kerülni sok munkával járt. 1961-ben fejlesztők egy csoportja Angliában, Manchesterben tervezett egy módszert az overlay folyamat automatikus végrehajtására, anélkül, hogy a programozó tudná, hogy mi történik (Fotheringham, 1961).Ez az eljárás, amit most VIRTUÁLIS MEMÓRIÁNAK hívnak, nyilvánvalóan előnyös, mert megkíméli a programozót egy csomó unalmas könyveléstől. Először 1960-as években használták számos gépen, leginkább számítógép rendszertervezéssel kapcsolatos kutatási tervekkel társítva. az 1970-es évek elejére a virtuális memória elérhetővé vált a legtöbb számítógép számára. Most még az egy chipes számítógépeknek, beleértve a Pentium II és UltraSPARC II, is van kifinomult virtuális memória rendszere. meg fogjuk ezt nézni később ebben a fejezetben.

6. 1. 1 LAPOZÁS

Az ötlet, amivel a Manchester csoport előállt az volt, hogy elválasztotta az adattároló hely fogalmát a memóriahelytől. Történetesen vegyünk példának egy olyan kor számítógépét, aminek az utasításai között esetleg 16-bites adattároló mezője lehet, és 4096 szavas memóriája. A programja 65536 memóriaszót tud tárolni. A 65536 (2 a 16-dikon) 16 bites adattárolás mind egy-egy külön memória szónak felel meg. De azért ne felejtsük el, hogy a tárolható szavak csak a tárolt bitek számától függés nem az aktuálisan elérhető memóriaszavak számától. Ennek a számítógépnek az adattároló helye a 0, 1, 2, …, 65535 számokat tartalmazza, mert annyi a lehetséges tároló hely. Ennek ellenére a gép memóriaszavainak száma valószínűleg kevesebb lenne, mint 65535.

A virtuális memória feltalálása előtt az embereknek különbséget kellett volna tenni a 4096 alatti, azzal egyenlő, illetve 4096 feletti adattárolók között. Habár ritkán használunk ennyi szót, ezt a két részt úgy tekintjük, mint a hasznos adattároló helyet és a használatlant (a 4096 feletti tárolókat használatlannak nevezzük, mert nincs összhangban az aktuális memóriatárolókkal). Régen nem tettek különbséget az adattároló hely és a memóriatároló között, mert a hardware egymásnak teljesen megfeleltetve kezelte a két dolgot.

A szétválasztásuk ötlete a következő. Rövid időn belül közvetlenül elérték a 4096 szavas memóriát, de nem volt szükséges, hogy összeegyeztethető legyen a 0-tól 4095-ig terjedő memóriatárolóval. Például “megmondhatjuk” a számítógépnek, hogy ezentúl amikor a 4096-os tárolóra utalunk, akkor a 0-s memóriaszót használja. Amikor a 4097-esre utalunk , akkor az 1-es memóriaszót használja, s mikor a 8191-esre utalunk, akkor a 4095-öst és így tovább. Más szóval végeztünk egy leképzést az adattároló az helyről aktuális memória tárolóra , amint azt a 6-2-es ábra mutatja.



6-2. ábra A leképzés ,amin keresztül a 4096-8191-es adattárolók vetítődnek a 0-4095-ös memóriatárba

Ezek alapján egy 4K-os gépnek, amely nem rendelkezik virtuális memóriával, egyszerű rögzített levetítése van a 0-4095-ös tárolók és a 4096 szavas memória között. Érdekes kérdés: "Mi történik, ha a program egy 8192 és 12287 közötti tárolóra ágazik?" Egy virtuális memória nélküli gépen a program hibát okozhat, ami a képernyőn a "Nemlétező memória előhívása" ("Nonexisting memory referenced") formában jelenik meg és a program leáll. Egy virtuális memóriával rendelkező gépnél a következő lépések jelennek meg:

1. A főmemória tartalmát elmenti a lemezre.
2. A 8192-12287-es szavakat lemezre helyezi.
3. A 8192-12287-es szavakat letölti a főmemóriába.
4. A tároló térkép megváltozik , és a 8192-12287-es tárolókat a 0-4095-ös memóriahelyre változtatja
5. A végrehajtás folytatódik, mintha mi sem történt volna.

Az automatikus fölé helyezésnek ezt a technikáját nevezzük lapozásnak, és a program nagyrészét, amit a lemezről betáplálunk , oldalaknak.

Létezik egy hamisabb módja is az adattárak memóriatárakba való leképezésének. Hangsúlyozásképpen: virtuális adattároló helynek nevezzük az adattárakat, amikre a program hivatkozik, és fizikai adattároló helynek a valóságos áramkörből álló memóriatárakat. A memóriatérkép (vagy oldaltábla) a virtuális adattárakat a pszichikai adattárakhoz tartozónak veszi. Tegyük fel, hogy van elég hely a lemezen ahhoz, hogy az egész virtuális adattárhelyet (vagy legalább is az általunk használt részét) rajta tároljuk.

A programokat úgy írják, mintha lenne elég főmemória az egész virtuális adattároló helynek, valójában azonban ez nem így van. A programok letölthetők (vagy tárolhatók) bármilyen szóról a virtuális adattároló helyeken, vagy elágazhat bármilyen, akárhol tárolt utasításra a virtuális adattároló helyen belül, tekintet nélkül arra, hogy igazából nincs elég pszichikai memória. Valójában a programozó nem írhat programokat anélkül, hogy tudatában ne lenne, hogy a virtuális memória létezik. A számítógép egyébként úgy néz ki, mintha hatalmas memóriája lenne.

Ez kritikus pont és később szegmentációkkal elemezzük, ahol is a programozónak tudatában kell lennie a szegmensek létezésével. Még egyszer hangsúlyozzuk, a lapozás a programozónak egy nagy, folyamatos lineáris memória illúzióját kelti, aminek mérete megegyezik a virtuális adattároló helyével. A valóságban elérhető főmemória kisebb (vagy nagyobb) lehet , mint a virtuális adattároló hely. E lapozással elért nagy főmemória létezésének tettetése programmal nem kinyomozható.(Kivéve , ha lefuttatjuk az időzített teszteket.) Amikor egy adattárolóra utalunk, a pontos utasítás , vagy adatszó jelenik meg. S mert a programozó programozhat úgy, mintha a lapozás nem létezne, a lapozás mechanizmusát átlátszónak mondjuk.

Ezek után már nem új az az ötlet, hogy a programozó úgy is használhat bizonyos nemlétező tulajdonságokat, hogy érdekelné hogy működik. Az ISA- szintű utasításcsomagok gyakran tartalmaznak egy MUL utasítást, még akkor is ha az alatta fekvő mikroarchitektúrának hardware-szinten van egy megsokszorozott terve. Az illúziót, hogy a gép tud szorozni tipikusan a mikrokód tartja fenn. Hasonlóan a virtuális gép is, ami operációs rendszerrel van ellátva, azt az illúziót keltheti, hogy az összes virtuális adattárat újraépíti a valós memória, pedig ez nem igaz. Csak az operációs rendszer íróinak (és a programozó diákoknak) kell tudniuk, hogy hogy

működik ez az illúzió a háttérben.

6.1.2. A lapozás megvalósítása

Egy fontos követelmény a virtuális memóriával kapcsolatban, hogy legyen egy lemez, amin a teljes programot és az összes adatot tárolhatjuk. Az ötlet egyszerűbb, ha a program lemezre történő másolatára úgy tekintünk, mint az eredetire; a főmemóriába időről-időre előhozott részekre sem másképp, mint másolatokra. Természetesen fontos , hogy az eredeti naprakészséget megtartsuk. Amikor a főmemóriában változtatunk, akkor annak tükröződnie kell az eredetiben is.
A virtuális adattároló hely számos azonos méretű részre bomlik. A rész mérete jelenleg az 512 byte-tól 64 Kbyte-ig terjed általában, bár alkalmanként 4MB-os méretűeket is használnak. A rész mérete mindig a kettes számrendszertől függ. A fizikai adattároló hely részekre bomlása azonos módon történik ,minden rész oldalnyi méretű lesz, azaz minden egyes főmemóriarész…